Automating Bug Report Assignment

John Anvik
Department of Computer Science
University of British Columbia
janvik@cs.ubc.ca

ABSTRACT

Open-source development projects typically support an open bug repository to which both developers and users can report bugs. A report that appears 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. The thesis of this work is that the task of triage can be eased by using a semi-automated approach to assign bug reports to developers. The approach consists of constructing a recommender for bug assignments; examined are both a range of algorithms that can be used and the various kinds of information provided to the algorithms. The proposed work seeks to determine through human experimentation a sufficient level of precision for the recommendations, and to analytically determine the tradeoffs of the various algorithmic and information choices.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Management

Keywords

Bug report assignment, triage

1. THE BUG TRIAGE PROBLEM

> "Given enough eyeballs, all bugs are shallow." I dub this: "Linus' Law". – Eric Raymond [11]

Open-source software projects commonly use an open bug repository to allow both developers and users to post problems encountered with the software, suggest possible enhancements, and comment upon existing bug reports. A 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 [11].

However, this potential advantage 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 [12]. Consider the case of the Eclipse Platform.[1] Figure 1 shows the rate of bug report submission for the Eclipse Platform for the one year period between the release of version 3.0 and 3.1. During this period 13,016 reports were filed, averaging 37 reports per day, with a maximum of 220 reports in a single day. Assuming that a triager takes approximately five minutes to read and handle each report, three person-hours per day was spent on average triaging bug reports.[2]

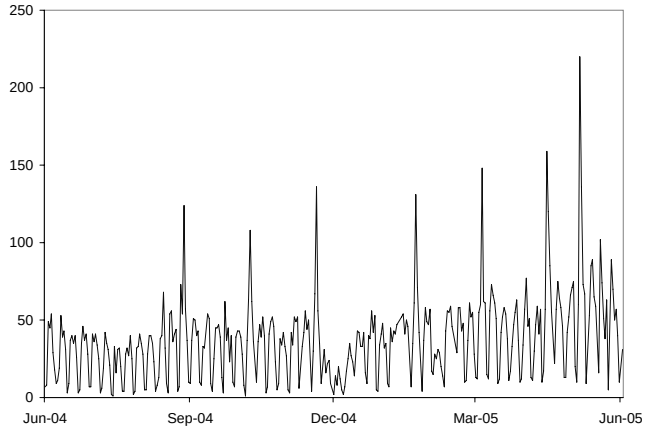

Figure 1: Daily rate of bugs submitted to Eclipse Platform project over one year.

The triage problem is not exclusive to open-source software projects. Members of closed-source projects, whose bug repositories are not openly available, have commented on the same problem. However, quantitative evidence of the magnitude of the problem is not available.

1.1 Bug Triage: State of the Practice

The current state of the practice for bug triage is to use a manual approach, and the approach differs from project to project. For the Mozilla project bug reports are triaged by quality assurance volunteers, rather than the developers, because of the volume of reports. A triager from the project

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

[2]The average time taken to triage bug reports is not known.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

commented:

> Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle.[3]

Early on, the Eclipse Platform 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.[4]

1.2 Automating Bug Triage

Regardless of the specifics 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. We would like to recover as much of these resources as possible by minimizing the need to have a person triage the bug reports. Fully automating the triage process may not be a realistic goal because some human interaction is necessary due to the amount of contextual knowledge required to correctly make decisions about each bug report.

1.2.1 Semi-automating Bug Report Assignment

A significant number of bug reports in bug repositories describe valid problems [1]. As all of these reports must be assigned to a developer, bug report assignment is an important part of triaging bug reports. Bug assignment is also important because mistakes made in assignment cause delays in the resolution of a bug report.

The assignment of bug reports is an example of a triage task that requires a lot of contextual knowledge. The triager needs to draw on knowledge about the product, the development team structure, individual developer experience and expertise, who handled similar bug reports, development schedules, and other information relevant for a particular bug report.

We believe that *semi-automation of bug assignment can improve the bug fixing process by reducing the average time taken to triage a bug and by reducing the number of incorrect assignments made by triagers.*

As bug assignment has similar characteristics to those of other bug triage activities, the semi-automation of this task provides a model for semi-automating other triage tasks. It is also an example of a human-supported task that is embedded in a process, which raises the question of how much automation is sufficient and how much incremental improvement can be made by using more information.

2. ANATOMY OF A BUG REPORT

Bug reports stored in open bug repositories such as Bugzilla,[5] GNATS,[6] and JIRA[7] all have a similar structure. 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.

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.

Bug 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 (i.e. bug dependencies) and the activity of each bug report. The activity log provides a historical record of how the report has changed over time, such as when the report has been reassigned, or when its priority has been changed.

3. PROPOSED SOLUTION

Our proposed solution is to create a recommender that produces a set of possible developers to whom a bug report might reasonably be assigned. The recommender for a project is created by providing an algorithm with information about previously fixed bug reports to create a model of expertise of the project developers. This model is then used to provide the set of recommendations.

More specifically, a recommender is created for a project by providing instances of information types ($I_1 \ldots I_n$), such as the bug report description, to a recommendation algorithm ($A_1 \ldots A_n$) as shown in Figure 2. The recommender is used then to produce a set of recommendations ($R_1 \ldots R_n$) by providing information about the new bug report to the recommender (see Figure 3).

Although there are many possible recommendation algorithms, our work is restricted to examining machine learning algorithms. The algorithms that we examine are supervised machine learning algorithms, clustering algorithms, and expertise networks. Just as there are many possible recommendation algorithms, there are many possible information types that could be used. Our work examines eight information types:

1. the textual description of the bug,
2. the component the bug is being reported for,
3. the operating system that the bug occurs on,
4. the hardware that the bug occurs on,
5. the version of the software the bug was observed for,
6. the developer who owns the associated code,
7. the current workload of the developers,
8. a list of developers actively contributing to the project.

[3]Personal communication with M.W., 05/03/05
[4]Personal communication with D.H., 23/02/05
[5]`www.bugzilla.org/`, verified 26/08/05
[6]`www.gnu.org/software/gnats/`, verified 07/09/05
[7]`www.atlassian.com/software/jira/`, verified 07/09/05

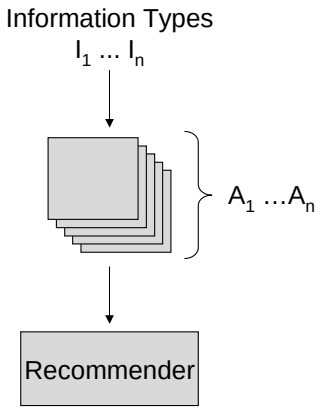

Figure 2: Creating a recommender.

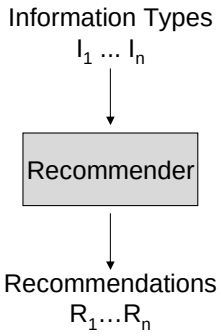

Figure 3: Making a recommendation.

With the exception of code ownership, all this information can be extracted from bug reports. The code ownership may be found by correlating information from a bug report, such as an exception trace, with an ownership architecture [3].

Given that sufficient information is available, the precision of a recommender is a function of the recommendation algorithm and the information types used. This precision lies along a spectrum as shown at the bottom of Figure 4. There exists a point along this spectrum at which the recommender is sufficiently precise to be of practical use by triagers. The exact location of this point is unknown, but will be a topic of investigation.

3.1 Finding the Point

To date we have evaluated nine instances of our approach framework (four supervised machine learning algorithms and one clustering algorithm with different combinations of three information types) and analytically compared their precisions [2]. From this previous work, we found that using a Support Vector Machine algorithm [6, 8] with the bug description, product component, and a list of actively contributing developers produces a recommender with 64% precision for Firefox and 86% precision for Eclipse. We believe that this is sufficient precision for an evaluation of the recommenders by Mozilla and Eclipse triagers. In Figure 4 this point is represented by the point labeled $A_3(I_1,I_2,I_3)$.

To determine if we have reached the point of sufficient precision, we plan to conduct a human evaluation of our approach. To aid this evaluation, a Firefox extension which presents recommendations for the currently viewed bug report has been created for use by Mozilla triagers. A similar Eclipse plug-in will be created for use by Eclipse triagers.

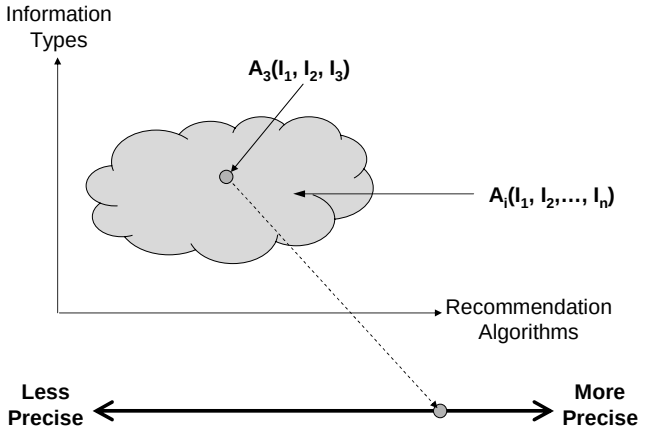

Figure 4: The precision space for automated bug report assignment.

The triager evaluation will be done in two phases. The first phase will collect baseline data that will be used to gauge the effect of the assignment recommender. During this time period two metrics will be collected: the time it takes for a triager to triage a bug report, and which reports they triage.

During the second phase of the evaluation, the same metrics from phase one will be collected, as well as information such as how often the triager used a recommendation, and the perceived usefulness of the tool.

Comparing the average time to triage a bug report during each of these phases will show whether or not we have reduced the average time to triage a report, and comparing the number of reports that are reassigned before and after the use of the tool will show if the tool helped to reduce the number of reassignments.

3.2 Further Exploration of the Precision Space

In addition to the human evaluation of our bug assignment recommender, we will continue to explore the precision space (the cloud labeled $A_i(I_1,I_2, \ldots,I_n)$ in Figure 4).

This space is occupied by other combinations of recommendation algorithms and information types. As mentioned in Section 3.1, we have already explored some of this space using three supervised machine learning algorithms, a clustering algorithm, and three information types. The remaining five information types from Section 3 will be explored using an information theoretic approach to feature selection [7] to evaluate their contribution to improving the precision of a recommender. In addition, two additional recommendation algorithms will be evaluated: a nearest-neighbour clustering algorithm [13] and an expertise network. The expertise network is to be a graph that has two types of nodes: experience atoms [9], representing data from the information types and developer nodes, representing individual developers. Edges weighted by atom frequency will connect experience atoms to developers. Recommendations are made by summing the weighted edges of the relevant experience atoms to provide

a developer score for a new report. Ranking these scores provides the list of recommendations for a new bug report.

These additional combinations of recommendation algorithms and information types will be evaluated analytically and compared to the results of our previous work [2] to determine what decision-making mechanisms and information types improve or degrade the precision.

4. RELATED WORK

We are aware of only two other efforts in automated bug assignment. Čubranić and Murphy [5] used a text categorization approach similar to our previous work [2], and with a Naïve Bayes recommendation algorithm achieved precision levels of around 30% on Eclipse. Our work 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 [4] outline an approach based on information retrieval in which they reported recall levels of around 20% for Mozilla. This work presents an approach that achieves a higher level of precision for these two projects, and we believe it is likely to help triagers on more systems.

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.

5. EXPECTED CONTRIBUTIONS

This research is expected to make three contributions:

1. A demonstration of an approach to semi-automate bug assignment that improves the bug fixing process by making it easier to find an appropriate developer and by reducing the number of incorrect assignments.

2. A characterization of the space of improvements to the approach using different recommendation algorithms and the contribution of each information type in creating a recommender for bug assignment.

3. A methodology for creating semi-automated approaches for bug triage tasks.

6. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of Eclipse Technology Exchange Workshop (eTX) at OOPSLA 2005*, pages 39–43, 2005.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, 2006. To appear.

[3] 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.

[4] 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.

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

[6] S. R. Gunn. Support Vector Machines for classification and regression. Technical report, University of Southampton, 1998.

[7] I. Guyon and A. Elissee. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[8] 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.

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

[12] 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.

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