Assisting Bug Report Triage through Recommendation

by

John Karsten Anvik

M.S., University of Alberta, 2002

B.Sc., University of Victoria, 2000

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

November 2007

Abstract

A key collaborative hub for many software development projects is the issue tracking system, or bug repository. The use of a bug repository can improve the software development process in a number of ways including allowing developers who are geographically distributed to communicate about project development. However, reports added to the repository need to be triaged by a human, called the triager, to determine if reports are meaningful. If a report is meaningful, the triager decides how to organize the report for integration into the project's development process. We call triager decisions with the goal of determining if a report is meaningful, repository-oriented decisions, and triager decisions that organize reports for the development process, development-oriented decisions.

Triagers can become overwhelmed by the number of reports added to the repository. Time spent triaging also typically diverts valuable resources away from the improvement of the product to the managing of the development process. To assist triagers, this dissertation presents a machine learning approach to create recommenders that assist with a variety of development-oriented decisions. In this way, we strive to reduce human involvement in triage by moving the triager's role from having to gather information to make a decision to that of confirming a suggestion.

This dissertation introduces a triage-assisting recommender creation process that can create a variety of different development-oriented decision recommenders for a range of projects. The recommenders created with this approach are accurate: recommenders for which developer to assign a report have a precision of 70% to 98% over five open source projects, recommenders for which product component the report is for have a recall of 72% to 92%, and recommenders for who to add to the cc: list of a report that have a recall of 46% to 72%. We have evaluated recommenders created with our triage-assisting recommender creation process using both an analytic evaluation and a field study. In addition, we present in this dissertation an approach to assist project members to specify the project-specific values for the triage-assisting recommender creation process, and show that such recommenders can be created with a subset of the repository data.

Contents

List of Tables

List of Figures

Acknowledgments

Just as it takes a village to raise a child, so too does it take many people playing both large and small roles to have work such as this come about:

1. Gail Murphy, my supervisor, without whom this work would never have gotten very far. Gail was helpful in a number of respect. First, she helped me frequently take a step back and look at things in a new light after having my nose buried in the details for far too long. Second, she alerted me to when my writing became too "encoded", admittedly a common occurrence. Lastly, she helped me through juggling this work and my family responsibilities, most often simply with an attentive ear.

2. Danielle, my wife, who juggled three, and then four, children at home after I left each morning and allowed me to focus on this work. I could not have accomplished this work without her unfailing support.

3. Bjorn Freeman-Benson, Eric Wohlstadter and Kelly Booth, my supervisory committee who helped me to take a broader view of this work.

4. Cristina Conati and Carson Woo, the university examiners, and Prem Devanbu, the external examiner, who gave constructive comments that greatly improved the quality of this dissertation. It was also Cristina's course that provided some of the initial spark for this work and the course project from which this work grew.

5. Davor Čubranić, whose paper also provided some of the initial spark and helped to shape this work.

6. The Eclipse Platform UI triagers who participated in the field study of the recommenders and gave data and feedback on how the recommenders worked in practice.

7. Randy Vance, my brother-in-law, who fed me and let me sleep on his floor for weeks so that I could finish writing this dissertation after moving my family to Victoria.

8. Chris Dutchyn, Andrew Eisenberg, Thomas Fritz, Lyndon Hiew, Terry Hon, Jonathan Sillito, David Shepherd and others from the Software Practices Lab who provided advice, support and distraction over the course of this work.

For Tara, John, Erik, and Kira.

Chapter 1

Introduction

A key collaborative hub for many software projects is a database of reports describing both bugs that need to be fixed and new features to be added [18][1]. This database is often called a *bug repository*[2] or issue tracking system. The use of a bug repository can improve the development process in a number of ways: it allows the evolution of the project to be tracked by knowing how many reports[3] are outstanding [15, 20], it allows developers who are geographically distributed to communicate about project development [46, 57], it enables approaches to determine which developers have expertise in different areas of the product [5], it can help improve the quality of the software produced [17, 46, 51], and it can provide visibility to users about the status of problem reports.[4] The bug repository can thus provide a location for users, developers, quality assurance teams and managers to engage in a "user-integrated development process" [30, page 4].

However, the use of a bug repository also has a cost. Developers can become overwhelmed with the number of reports submitted to the bug repository as each report needs to be *triaged*[5] [39, 52]. Each report is triaged to determine if it describes a valid problem and if so, how the report should be categorized for handling through the development process [23, 52]. When developers are overwhelmed by reports, there are two effects. The first is that effort is redirected away from improving the product to managing the project. If a project gets thirty reports a day and it takes five minutes to triage a report,[6] then over two-person hours per day are spent triaging reports. If all of these reports led to improvements

[1]Also see `http://wiki.mozilla.org/MozillaQualityAssurance:Triage`, verified 01/08/07.

[2]This name is obviously a misnomer, the repository does not contain the bugs, just descriptions of the bugs.

[3]We will use both the term *report* and the more colloquial *bug report* interchangeably to refer the items in a bug repository, as the repository typically contains both fault reports and feature requests.

[4]As data is more readily accessible about and from bug repositories used in open source development projects, we focus our comments and examples on repositories from this type of project. However, we believe that the issues discussed in this dissertation are applicable to bug repositories from both open and closed source development projects.

[5]triage: A process in which things are ranked in terms of importance or priority. (Merriam-Webster Dictionary)

[6]These numbers are based on our observation of an open-source project.

in the code, this might be an acceptable cost to the project. However, for some projects, less than half of submitted reports lead to code improvements. For example, we found that the Eclipse[7] project had 5515 unproductive reports in 2004 [3].

The second effect is that reports may not be addressed in a timely fashion. If the number of reports that enter the repository is more than can be reasonably triaged within a suitable amount of time for the project, then some reports may languish in the repository as other reports demanding more immediate attention take precedence. For an open-source project where the responsiveness of the development team to the community is often measured by how quickly reports are addressed and the number of outstanding reports, the rate at which reports are triaged can be an important factor in how the project is perceived. For example, Crowston et al. found that a measure of success for an open source project is the rate that users submitted bug reports and participated in project mailing lists [17].

The person who triages the report, the *triager*, should have two goals. The first goal is to have the repository contain the smallest set of best reports for the project. The smallest set of best reports is desirable because reports typically enter the repository from a variety of sources, such as members of a technical support division, other developers, and the user community. Unfortunately, with so many different sources of reports, some of the reports are not meaningful [3]. For example, on a large project with many team members, several developers may submit a report describing the same bug [30]. These duplicate reports need to be gathered together so that development effort is not wasted by having two developers solve the same problem [32, 55]. A triager also needs to filter reports that do not adequately enable a bug to be reproduced or that describe a problem whose cause is not the product, but rather is something beyond the control of the developers, such as the operating system. Sometimes, a triager also needs to filter out reports that are spam. Finally, a triager may indicate that the problem will not be fixed or that the feature will not be added to the product. Reports meeting any of these criteria must be identified so that development effort can focus on the reports that lead to product improvements. For example, nearly a third of the reports submitted to the Firefox[8] project created between May 2003 and August 2005 were marked as duplicates [3]. We call triage decisions that result in a report being designated as not meaningful as a *repository-oriented decisions.*

A second goal of the triager is to organize the reports for integration into the development process. Reports may be organized in a variety of ways. For example, a report may be categorized by the product component[9] it effects so that the report is routed to

[7]Eclipse provides an extensible development environment, including a Java IDE, and can be found at `www.eclipse.org` (verified 06/06/07).

[8]Firefox is a web browser and can be found at `www.mozilla.org/products/firefox` (verified 06/06/07).

[9]We use the term component to refer to a specific piece of product functionality, such as the user interface, network protocols, or business logic.

the development team responsible for that component. Alternatively, a report may be categorized by developer according to who has the expertise to resolve it; in other words, the report is assigned to a developer. Finally, a report may be categorized by other project members that may want to be informed of progress on the report. We call these types of decisions *development-oriented decisions.*

Typically, development-oriented decisions are made after repository-oriented decisions. However, the two decisions may be intermixed. For example, it may be that a triager lacks the knowledge to determine if a report duplicates another report (a repository-oriented decision). The triager may choose instead to assign the report to a developer (a development-oriented decision) so that an appropriate developer can decide whether or not the report describes a new problem.

Every repository-oriented or development-oriented decision that must be made has a cost. One source of cost is the time and effort required to gather the information to make the decision. For example, to determine if a report duplicates a report that already exists in the repository, the triager must search the repository based on keywords and examine the results for potential duplicates. As another example, to make a decision about which developer should be assigned the report, the triager needs to gather information about who has the expertise to fix the problem. The lowest cost means to gather the information is from the triager's own experience. When a decision lies outside that experience, more expensive approaches must be used, such as consulting a project web site that states the responsibilities of developers or searching for similar reports in the repository. For example, we estimate that the Eclipse project spent 450 person-hours on unproductive reports in 2004 (5515 reports $\times$ 5 minutes / report) [3].

This dissertation presents an approach to assist triagers with the development-oriented decisions they make during triage activities. The goal behind this approach is to lower the cost of triage activities by reducing human involvement in the triage process. Our approach uses a supervised machine-learning algorithm to build models for different report categories (e.g., by component or by developer). The models are then used to provide the triager with recommendations for a new report that is submitted to the repository. In this way, we strive to reduce human involvement in triage by moving the triager's role from having to gather information to make a decision to that of confirming a suggestion. The thesis of this work is that

> *human involvement in the bug triage process can be reduced using recommenders; these recommenders can be created using a prescribed process we call the triage-assisting recommender creation process.*

The rest of this chapter proceeds as follows. First, we provide a brief overview of bug reports, followed by a description of the triage process. We then describe the triage-assisting recommender creation process. We conclude by outlining previous work and the contributions of this work.

1.1 An Overview of Bug Reports

A bug report contains a variety of information. Some of the information is categorical such as the report's identification number, its resolution status (i.e., new, unconfirmed, resolved), the product component the report is believed to involve and which developer has been given responsibility for the report. Other information is descriptive, such as the title of the report, the description of the report and additional comments, such as discussions about possible approaches to resolving the report. Finally, the report may have other information, such as attachments or a list of reports that need to be addressed before this report can be resolved. A more detailed description of bug reports is given in Section 2.2.

1.2 The Bug Triage Process

Everyday, almost 300 bugs appear that need triaging. This [number of reports] is far too much for only the Mozilla programmers to handle.[10]

As previously mentioned, the decisions that a triager makes can be divided into two types: repository-oriented or development-oriented. These types of decisions can be seen in the description of a triager's responsibilities for the Gnome project:[11]

1. "Making sure the bug has not already been reported before."
2. "Making sure the bug has enough information for the developers and makes sense."
3. "Making sure the bug is filed in the correct place."
4. "Making sure the bug has sensible "Severity" and "Priority" fields."
5. "Making sure the bug is versioned correctly." [10]

The first two decisions listed are repository-oriented decisions. Their intent is to remove reports that will not contribute to the overall improvement of the product. The remaining three decisions are development-oriented.

1.2.1 A Walkthrough of the Triage Process

This section presents a picture how triage is generally performed. To create this picture, we gathered information from a variety of sources including documentation from open source

[10]Personal communication with Mozilla developer, 05/03/05

[11]The Gnome project provides a graphical desktop for the Linux or UNIX operating system. See www.gnome.org, verified 09/08/07.

project websites, observations derived from the comments and history log of bug reports, and email correspondence and interviews with triagers.

A triager begins their duty by searching the repository for newly submitted reports for the product component of interest. For convenience, triagers typically use a repository query that returns such a list. Having obtained a list of reports to triage, the triager will select a report and then read the title and description. If the description of the problem is not clear, the triager will typically add a comment to the report asking for clarification and for some projects change the report status to indicate that more information has been requested.

The triager's first responsibility is to make the repository-oriented decisions. Typically, the first repository-oriented decision that the triager makes is determining if the problem is already described in the repository (i.e., the new report is a duplicate of the existing report in the repository). To determine if the report is a duplicate, the triager will first rely on their project knowledge. A trivial example occurs when a reporter inadvertently submitted the exact same report twice and the triager has just triaged the first instance. The triager also often uses keyword and text searches of the repository to find reports that describe the same problem.

If the report describes a defect, and the triager gains confidence that the defect has not been previously reported, the triager will next try to reproduce the problem. If he is unable to reproduce the problem, then the triager will indicate that the problem could not be confirmed and indicate this information by changing the status of the report. If the problem is confirmed, the triager will sometimes attach a test case to help the developer in correcting the problem.

Having made repository-oriented decisions such as establishing that the report describes a new problem or desired functionality, or that the defect is reproducible, the triager will begin to make development-oriented decisions. In making the repository-oriented decisions, the triager may have gathered enough information to combine with their own project knowledge to make such decisions as how to categorize the report with respect to project and to whom to assign the report. If the triager has not yet gathered enough information, they will either continue to search and examine reports in the repository to make these decisions or seek the advice of more experienced triagers and developers. For some projects, the triager will add specific keywords to assist other triagers and developers in knowing certain features about the problem or requested functionality, such as if the problem results in a crash or causes the application to freeze, or the type of functionality requested such as a modification to the user interface.

1.3 Recommender Creation Process Overview

The reports within a repository provide a wealth of information about a project's development process, especially about how reports are organized. Information relevant to report categorization can be extracted from a set of reports that have been previously categorized and used to predict how a new report should be categorized [4, 14, 19, 23]. For example, by extracting information from reports that were organized as belonging to the User Interface component, a model of user interface reports could be built and used to suggest if a new report belongs to this product component [23]. Similarly, information extracted from reports resolved by a particular developer can be used to recommend if a new report should be assigned to that developer. Such models can be built using a machine learning algorithm [4, 14, 19]. We refer to models that are created for the purpose of providing suggestions or recommendations as *recommenders.* We use the symbol R_{DO} to refer to the group of recommenders that assist with development-oriented decisions during bug triage. We can create separate recommenders for different categorization tasks using a similar process. For instance, a recommender that assists with component categorization is an example of a R_{DO} recommender. A recommender that assists with assigning reports to developers is another instance of a R_{DO} recommender.

In general, using a machine learning approach requires answering questions about what data to use, how to prepare the data, and what machine learning algorithm to use. To create a R_{DO} recommender, six questions need to be answered:

1. Which reports from the repository should be used?

2. How many reports should be used?

3. How should the reports be labeled?

4. What are the valid labels?

5. Which pieces of data from the reports should be used?

6. Which machine learning algorithm should be used?

Through our work, we have determined that half of the answers to these questions are the same regardless of the particular R_{DO}. The answers to the three questions of which reports to use, what pieces of data to use, and what machine learning algorithm to use are the same for each type[12] of R_{DO} we have investigated. Specifically, we have found that the use of textual information about a problem or enhancement of resolved reports[13] and the Support Vector Machines algorithm [33] can be used to create useful R_{DO}. The

[12]The type of a recommender refers to the information that the recommender recommends (e.g., a developer assignment recommender (R_A) is a type of R_{DO}). An instance of a R_{DO} refers to a type of R_{DO} that has been created for a specific project, such as a R_A for Eclipse.

[13]A report is considered resolved if it has reached the end of its life-cycle.

variation between the different recommenders is in how many reports are used, how the reports are labeled and which are the valid labels. For brevity, we refer to the machine learning-based triage-assisting recommender creation process — our approach for creating a R_{DO} recommender — as ML_{Triage}.

For example, consider creating a recommender to help with assigning new bug reports (an assignment recommender or R_A) for the Eclipse Platform project. To use ML_{Triage}, we begin by collecting resolved reports from the most recent eight months (i.e., answer the questions of which reports and how many reports to use). We then label the reports with the name of the developer who resolved the report (i.e., answer the question of how to label the reports). As developers may have left the project in the last eight months, we only keep reports that are labeled by the current Eclipse developers (i.e., answer the question of which labels are valid). Having collected the reports we will use for training the R_A, we extract the summary and description from each report. This textual information, and the label of the report it came from, are then given as input to a Support Vector Machines supervised machine learning algorithm [31]. After the algorithm has created the R_A, we can provide the recommender the textual information of a new report and receive back a list of recommendations that can be presented to help the human triager.

To compare the performance of different types and configurations of R_{DO} we use the traditional measures of precision, recall, and F-measure. To enable experimentation with different recommenders created using ML_{Triage}, we integrated four types of R_{DO} into a web tool for the Bugzilla issue tracking system.[14] The four types of R_{DO} we integrated were a developer assignment recommender (R_A), a component recommender (R_C), a sub-component recommender (R_S) and an interest recommender (R_I). We call our tool Sibyl.[15]

1.4 Survey of Related Work

This dissertation presents an approach (ML_{Triage}) to creating recommenders that assist with development-oriented decisions. Our approach is unique in three respects.

The first is with respect to generality. Previous approaches have looked at assisting with one development-oriented decision, either developer assignment [14, 19] or component assignment [23]. In contrast, ML_{Triage} can create recommenders for both of these types of decisions, as well as others, such as a recommender for who should be added to the cc: list (see Section 3.4), for estimating the effort needed to fix a bug (see Section 6.3) or for suggesting which files touch to fix a fault or implement a feature (see Section 6.3).

Others have also considered using machine learning to assist with making development-oriented decisions [14, 19, 23]. Whereas the previous authors applied their process to

[14]The Bugzilla issue tracking system is commonly used by open source projects and can be found at `www.bugzilla.org` (verified 11/06/07)

[15]A *sibyl* is a prophetess of the ancient world.

one [19, 23] or two [14] projects, we show that ML_{Triage} applies to a variety of projects.[16]

Our approach is also unique with respect to its performance. Previous approaches for creating a developer recommender have achieved accuracies of 20% [14] and 30% [19]. Using ML_{Triage}, we create developer recommenders with accuracies in the range of 70% to 98% for five different projects. Previous work for creating a component recommender achieved an accuracy of 84% [23] for making one component recommendation. Although we do not achieve a comparably high accuracy for one recommendation (45%-66%), the previous work examined suggestions for eight components, whereas we examined projects with eleven to thirty-four components and what we believe to be a lower quality of data that is found in open source projects.

1.5 Contributions

This dissertation makes the following contributions to the field of software engineering:

1. An approach/framework to creating R_{DO} recommenders that:
    - Generalizes across types of R_{DO}.
    - Generalizes across software projects.
    - Produces recommenders that are accurate.
2. A field study showing how R_{DO} recommenders work in practice.
3. An approach to assist project members in configuring project-specific parameters when creating a R_{DO} recommender that allows recommender creation to occur on a client instead of a server.

1.6 Organization of the Dissertation

This chapter presented one of the challenges faced by projects that use a bug repository: developers can become overwhelmed by the number of reports submitted to the repository. This challenge leads to the need to triage the reports. We presented an overview of the triage process and divided the decisions made by triagers into two categories: repository-oriented and development-oriented. This dissertation presents an approach (ML_{Triage}) that creates recommenders that provide suggestions for development-oriented decisions (R_{DO}).

In the next chapter, we provide a more detailed background of bug reports and machine learning in preparation for presenting the details of our approach for creating R_{DO} recommenders, ML_{Triage}. Following this, we describe how we tuned ML_{Triage} by creating a developer assignment recommender and then describe how ML_{Triage} is used

[16]Similar to previous work, we examine open source projects, as their process data is more readily accessible. However, we believe that ML_{Triage} also generalizes to closed source projects.

to create recommenders for a report's component field and cc: list (see Chapter 3). The choices made in tuning ML_{Triage} are evaluated in Chapter 4. Although some of the answers for ML_{Triage} are uniform across projects, others are project-specific. To assist a triager in configuring ML_{Triage} for a specific project, we present an assisted configuration approach for ML_{Triage} in Chapter 5. We then discuss various questions raised by this work (Chapter 6) and review related work (Chapter 7), before making our concluding remarks.

Chapter 2

Background

This chapter gives additional information on the triage process and provides background information on bug reports and machine learning algorithms. The additional information on the triage process was gathered from triager questionnaires, interviews, and project web sites and demonstrates how triage practice differs from project to project.

2.1 The Bug Triage Process

A general overview of the triage process was presented in Section 1.2. This section provides some further background into how the triage differs from project to project, and some of the challenges of the triage process.

2.1.1 Project Differences for the Triage Process

Although the types of decisions a triager must make generally do not vary from project to project, who performs the triage activities does vary. For example, triage for the Eclipse project is done by project developers, whereas triage for the Mozilla projects[1], KDE[2], and Gnome are primarily done by volunteers. When developers from the project perform triage, they are able to draw on deep project knowledge that may allow them to make correct decisions more often. However, the cost to the project is higher as the time that developers spend doing triage is time not spent improving the product. In contrast, when volunteers are used to perform triage, the triagers are more likely to make errors that may cause delays in the improvement of the product. However the project is able to make better use of developer resources.[3]

Like many development processes, the triage process for a particular project is not static and can change over time. For example, early in the history of the Eclipse project, a single developer triaged the reports. However, as the product matured and the frequency

[1]The Mozilla project comprises many projects, such as the popular Firefox web browser and the Thunderbird email client, and which share the same development process and bug repository.

[2]KDE is a graphical desktop for Linux and Unix workstations; `www.kde.org`, verified 09/08/07.

[3]See `live.gnome.org/Bugsquad/TriageGuide`, verified 23/08/07.

of additions to the bug repository increased, the task became too overwhelming for a single person. Triage was then decentralized and each component team was made responsible for monitoring the repository for reports that were categorized by their component.[4]

Assignment of Reports

We have observed that projects use one of two approaches for developer assignment: triager-directed assignment or self-assignment. For example, the Eclipse project used triager-directed assignment where the triager has the authority to direct reports to specific developers. In contrast, the Mozilla project uses self-assignment where reports are assigned to a default user name, typically associated with a specific project component, and then developers choose the reports that they wish to resolve.[5] This practice is also followed in the FreeBSD project [25]. However, projects that use this approach have also been observed to periodically assign reports to developers to get them addressed. For the Gnome project, reports are assigned to a team leader or someone in charge of coordinating maintenance activities [23].

As with other development activities, individual triagers approach report assignment differently depending on their experience with the project. In interviews with four of the triagers from the Eclipse project, we found that more experienced triagers only need to read the report description to make assignment decisions, while others look for a stack trace to point them to the right component or may need to refer to a web page to know who has current responsibility for a particular component or sub-component.

2.1.2 Some Challenges of the Triage Process

In interviews with four triagers for the Eclipse project, the triagers identified several challenges. A common challenge expressed by each triager was reproducing the problem described by the report. Other work has also made this observation [39]. One triager indicated that it was challenging to know about every area of the project and that they needed to build up additional project knowledge in order to triage more effectively.

Several of the triagers commented that the difficulty of triaging a report depended on the reporter. One triager commented that reports were easier to triage from individuals in whom he had confidence, either because the triager knows the individual, or because the individual is on a product component development team. Another triager categorized reporters into either "I'm using it" reporters or "I'm building on it" reporters. The first group generates reports that are hard to deal with as they often report the component wrong initially and do not define or describe the problem very well. The other type of report submitter, the "I'm building on it" group, typically produce reports that have good

[4]Personal communication with Eclipse developer, 23/02/05.

[5]Personal communication with Mozilla developers, 05/03/07. Also see `http://wiki.mozilla.org/MozillaQualityAssurance:Triage`, verified 01/08/07.

descriptions and are very technical, sometimes pointing out the code that was causing the problem. Triagers report that it is easier to triage reports from the second group than reports from the first group.

2.2 Bug Reports

To assist triagers in making development-oriented decisions, we create recommenders based on data found in bug reports. To provide more background on the data source that we use for making recommendations, this section provides on overview of the bug report. We present two aspects of a bug report: its anatomy and its life-cycle.

2.2.1 Anatomy of a Bug Report

Figure 2.1 shows an example of a report for the Eclipse project. The Eclipse project uses the Bugzilla bug repository software. Although we focus on the contents of a Bugzilla bug report in this section, bugs reports in other bug repositories such as JIRA,[6] GNATS,[7] and Trac[8] contain similar data. Each report includes pre-defined fields, free-form text, attachments, and dependencies.

The pre-defined fields provide a variety of categorical data about the 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 added, 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. These fields, with exception of those that are fixed at report creation, represent the different ways that a bug report can be categorized in the development process.

The free-form text includes the title of the report, a full description of the bug, and additional comments. The full description area 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 report. The

[6]`www.atlassian.com/software/jira`, verified 09/08/07
[7]`www.gnu.org/software/gnats`, verified 09/08/07
[8]`trac.edgewall.org`, verified 09/08/07

Figure 2.1: An example of a bug report.

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.

2.2.2 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 report in a generic Bugzilla repository (see Figure 2.2). However, reports in other repositories have a similar life-cycle (see Figure 2.3). Projects may add or remove states to better fit a project's development process. However, we have observed that these modifications typically are just specializations of the default states. For example, the Gnome project adds an additional NEEDSINFO state that is a specialization of the NEW state.

When a report is submitted to the repository, its status is set to UNCONFIRMED. Once the problem described in the report has been verified, the report moves to the NEW state. 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 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.

2.3 Machine Learning Algorithms

The recommenders described in this thesis are built using a machine learning technique. The field of machine learning is concerned with the development of algorithms and techniques that allow computers to learn [45, 68]. Although, the exact definition of learning in the context of computer learning seems to be disputed, one definition put forward by Mitchell [45] states:

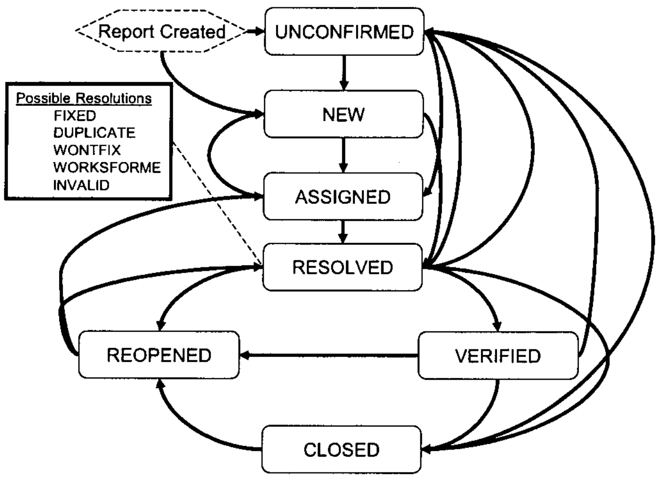


Figure 2.2: The life-cycle of a Bugzilla report.

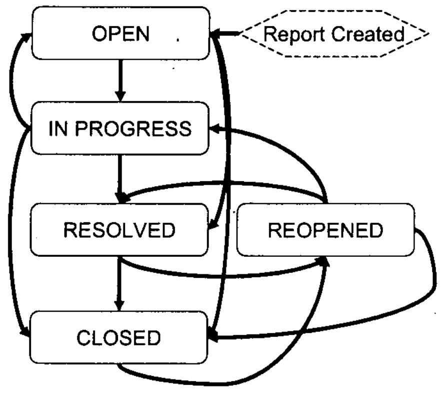


Figure 2.3: The life-cycle of a JIRA report.

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning algorithms fall under three categories: supervised learning, unsupervised learning, and reinforcement learning [45]. In this work, we investigate the use of different supervised learning algorithms to determine which algorithm is the most appropriate, as well as examining the use of one unsupervised learning algorithm. We do not examine the use of reinforcement learning as this form of machine learning focuses on learning as events happen, whereas we investigate learning from historical data.

Under the category of supervised learning, there are a number of potential algorithms that could be used for creating recommenders for assisted triage. These algorithms fall under five sub-categories: rule-based algorithms, probabilistic modeling algorithms, decision tree algorithms, non-linear modeling algorithms, and instance-based algorithms. We chose to investigate one instance of each of these categories: conjunctive rules [68], Naïve Bayes [34], C4.5 [50], Support Vector Machines [31], and nearest-neighbour [2] respectively. Since all unsupervised learning approaches are essentially a clustering algorithm at their core [68], we chose to consider one example only. The unsupervised learning algorithm that we chose was the clustering algorithm called Expectation Maximization [22].

We next present an overview of the algorithms we chose to evaluate and conclude this section with our reasoning for our choices.

2.3.1 Overview of Selected Machine Learning Algorithms

To understand how the various machine learning algorithms work, three concepts must be understood: the *attribute* (also called the *feature*), the *instance* and the *class*. An attribute is a specific piece of information that is used to determine the class, such as the component that a bug report is filed against or the name of the developer who fixed the fault described by the report. An instance is a collection of attributes that have specific values. For example, we may be creating instances that contain information about the component a report was filed against and the name of the developer who fixed it. The attributes of the instance are "Component" and "Fixed By". Table 2.1 shows examples of three instances. The first instance is for a report filed against the User Interface component and fixed by Tom. The second instance is for a File Processing report that was also fixed by Tom. The third instance is for a report for the User Interface component and fixed by Steve. Finally, a class is the collection of instances that all belong to the same category. For example, each row in Table 2.1 could belong to a particular project. From these instances, a recommender could be created that suggest if a bug report belongs to a specific project (i.e., the class) such as Firefox within the Mozilla bug repository. In this example, the attributes of "Component" and "Fixed By" are used for predicting the class Firefox. More

Table 2.1: Examples of instances used for creating a recommender.

	Component	Fixed By
Instance 1	User Interface	Tom
Instance 2	File Processing	Tom
Instance 3	User Interface	Steve

complex data, such as the text of the bug report, can also be used to provide the attributes for predicting the class of a bug report. Machine learning algorithms differ in how they use the values of the attributes to predict a class.

The rest of this subsection provides descriptions of the various algorithms we chose to evaluate.

Conjunctive Rules

Conjunctive rules is an instance of a rule-based approach [68]. The idea behind a rule-based approach is to determine a rule, or set of rules, for each class that describes all, or as many as possible, instances in that class.

The conjunctive rules algorithm consists of the creation of a set of rules, whereby each rule is the conjunction of *antecedents* that is associated with a particular class. A simple example of such a rule is "If the report is for the Mac OS X platform and is for the user interface component, then the report should be assigned to Keith".

Rules for each of the classes are constructed in the following manner. First all of the attributes for all of the instances for a class are gathered (i.e., all the features of the reports that have been labeled with Keith). Next, one of the attributes is selected as an antecedent based on its *information gain* [68]. In other words, the attribute is evaluated on how well it contributes towards determining that class of the instance. In this example, it means determining how well the attribute helps to determine that a report belongs to Keith as opposed to one of the other developers. A common term like the product name would not contribute very much information as it would likely appear in the reports labeled with other developer names, whereas the name of a specific component might contribute more information about which developer to assign the report.

The rule is built by progressively adding antecedents in the order of information gain until all of the attributes of the instances for the class have been used. As not all of the antecedents will be necessary for determining the class, the *reduced error pruning* technique is used to create the smallest rule possible for the class. Reduced error pruning works by iteratively removing antecedents and using a portion of the instances for the class to evaluate how well the pruned rule correctly classifies an instance. The set of instances used for evaluation is set aside prior to creating the rule. In other words, a rule is built using a subset of the reports labeled by Keith and the remaining reports are used to evaluate the rule. The rule is systematically pruned by removing report attributes with low information gain and the pruned rule is evaluated by how well the rule correctly classifies the reports that were set aside. Pruning continues until the error rate is minimized.

Once the pruned rules for the classes have been generated, they are applied to a new report to classify it.

Naïve Bayes

Naïve Bayes is a probabilistic technique that uses Bayes rule of conditional probability to determine the probability that an instance belongs to a certain class. For example, if the word *widget* occurs more frequently in the reports resolved by Tom than the reports resolved by Steve, then it is more likely that a new report that has the word *widget* in the description should be given to Tom to resolve.

Intuitively, Bayes rule states that if something occurred frequently in the past, then it is more likely to occur in the future (see Equation 2.1).

$$PosteriorProbability = \frac{PriorProbability * Likelihood}{Evidence} \tag{2.1}$$

More formally, Bayes rules states that the probability that an instance is of class C given that the instance I has certain feature values $f_1 \dots f_n$ (i.e., $p(C|f_1 \dots f_n)$) is the probability that the instance is of class C (i.e., $p(C)$) times the probability that the feature values will have these values if the class is C (i.e., $p(f_1 \dots f_n|C)$), all over the probability that the features will have the values of $f_1 \dots f_n$ (i.e., $p(f_1 \dots f_n)$) (see Equation 2.2).

$$p(C|f_1 \dots f_n) = \frac{p(C) \times p(f_1 \dots f_n|C)}{p(f_1 \dots f_n)} \tag{2.2}$$

The algorithm is called "Naive Bayes" as it makes the strong assumption of independence of the features. Although this assumption is false in many real-world applications, in practice recommenders based on Naïve Bayes perform surprisingly well and it has been found to not be a mathematically unreasonable assumption [26]. Taking the independence assumption into account reduces Equation 2.2 to Equation 2.3 where I represents the set of training instances:

$$p(C|f_1 \dots f_n) \propto \prod_{i=1}^{|n|} p(f_i|I) \tag{2.3}$$

In our use of the Naïve Bayes algorithm, the feature probabilities are based on the frequency of terms in the reports. However, this can lead to zero probabilities when a report does not have one of the terms. This problem, known as the zero-frequency problem [67], is commonly corrected using a Laplace estimator [68].

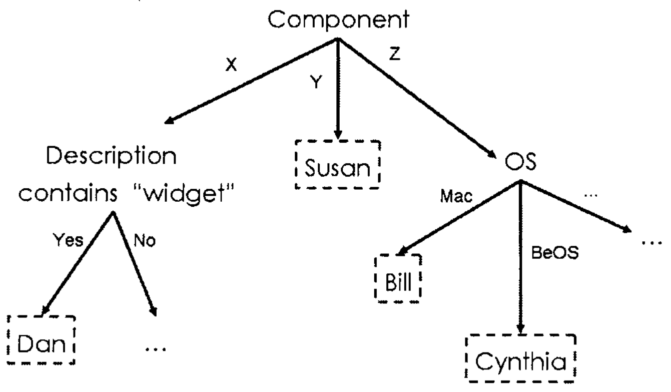


Figure 2.4: Example of a decision tree.

C4.5

The C4.5 algorithm [50] creates a decision tree based on the attributes of the instances in the training set. This algorithm differs from the others (with exception of conjunctive rules) in its inherent ability to model interaction between the attributes [23].

C4.5 begins by determining the *information gain* of each attribute and selecting the attribute that has the highest value. Information gain is a measure of how useful a particular attribute is in deciding between classes. An attribute with a high information gain will be good at discriminating between the different classes.

Having selected the attribute with the highest information gain (a_{best}), the C4.5 algorithm creates a decision node and then recurses on the remaining attributes obtained by splitting on a_{best}. The subtrees end in leaf nodes representing the classes. A predication is made by following the appropriate path through the decision tree based on the attribute values of the instance.

Figure 2.4 shows an example of the top of a decision tree created using C4.5 for deciding to which developer to assign a report. The figure shows that the C4.5 algorithm chose the Component attribute as the root of the tree. Depending on the value of Component, one of three paths is followed. If the Component is Y, then the report is assigned to Susan. If the Component is Z, then the next decision is made based on the value of the OS attribute, and so forth.

Support Vector Machines

Support Vector Machines is an algorithm that determines non-linear models describing each class and then uses these models to predict the class of new instances.

From the point-of-view of the Support Vector Machines algorithm, the different instances exist in a multi-dimensional space defined by the attributes. For example, if the instances have four attributes, then the instances are placed in a four-dimensional space according to their attribute values. SVM works by first determining the maximum margin

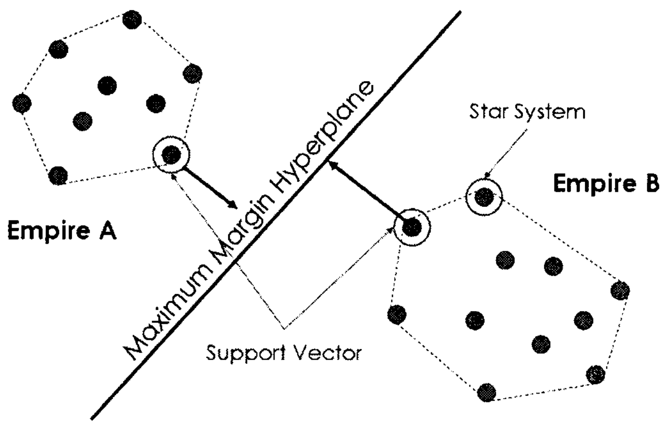


Figure 2.5: Support Vector Machines algorithm analogy.

hyper-plane(s) between the different classes in this multi-dimensional space. A hyper-plane is used to determine the support vectors, specific instances from each of the classes in the space. The support vectors are then used to create a non-linear equation describing each of the classes by setting the result of the equation to a specific value, such as one, and using the attributes values to determine the constants of the equation. To classify a new instance, the instances attribute values are plugged into the equations for each class and the largest result indicates the predicted class.

Consider the analogy demonstrated by Figure 2.5. There are two galactic empires, Empires A and B, that have agreed to a specific boundary in space (i.e., the hyper-plane). Each of the empires owns specific star systems (i.e., the instances) at certain spatial co-ordinates (i.e., the attributes). Since both empires want as much territory as possible, the boundary is such that does not come any closer to each of the two territories as is possible (i.e., the boundary is a maximum margin). To define this boundary, it is sufficient only to know which star systems are closest to that boundary (i.e., the support vectors). The co-ordinates of each of the star systems along this boundary are used to define an equation representing the territory of each empire. When a new star system is discovered, the equations are used to determine to which empire the star system belongs.

Expectation Maximization

Expectation Maximization is an iterative clustering algorithm that guesses at the probability that an instance belongs to a specific cluster (i.e., class) and then iteratively refines the guesses until a maximum log-likelihood of the probabilities is reached. The algorithm consists of two parts: guessing the probabilities (i.e., expectation) and maximizing the expected likelihood found in the previous step (i.e., maximization).

Imagine there is a set of reports that were resolved by two developers, but we do not know who resolved each report. As we do not know to which cluster (i.e., class) each report belongs, we are going to determine the probability that a report belongs to a certain cluster (i.e., was resolved by a particular developer). However, we do not know what the probability distribution is for either of the two clusters, so we estimate the parameters of the probability function of each cluster (i.e., the means and standard deviations) and sampling probability (i.e., the proportion of the reports that came from the first developer's group of reports). This estimation of parameters is the expectation step. We next use these estimated parameters to determine the probability that a report belongs to each developer and then compute the log likelihood for the parameters (i.e., the likelihood that we have found the right parameters). This step is the maximization step. We then use the computed class probabilities to estimate the parameters for the next iteration. We continue this cycle until the log likelihood has not increased more than a small amount, such as 10^{-10}, over a number of iterations. One way to view this process is that we place the reports into a single cluster and then tease apart the reports until they are in two clusters.

Nearest Neighbour

The nearest neighbour algorithm is an instance-based algorithm, also known as a lazy algorithm. As opposed to the other algorithms in which a function is determined for computing the probability that an instance belongs to a class, the nearest-neighbour algorithm uses all of the instances to make a prediction. The algorithm makes the prediction by comparing a new instance to all other instances (or representative instances) and finding the previous instance which is closest to it. The class of the new instances nearest neighbour is then used as the new instance's predicted class. The distance between two instances is found using a distance metric specific to the set of attributes.

2.3.2 Rationale of Choices

We chose to investigate rule-based algorithms and decision tree algorithms, because we believe that they bare some similarity to the reasoning used by triagers in making development-oriented decisions. For example, when deciding who to assign a report to the triager may use a rule such as "If the component is UI and the OS is Windows XP, then assign the report to Steve". Rule-based algorithms and decision tree algorithms bear some similarity. For example, a decision tree can be turned into a set of rules. However, they do have subtle differences. First, the set of rules generated by a rule-based algorithm tend to be clearer and less complex then the equivalent decision tree. Also, rules focus on defining a single rule that applies to one category without regard for the effect of the rule on other categories. In contrast, decision trees attempt to determine tests that distinguish between all classes. [68] Another perspective on these two algorithm types is that a decision tree is a top-down approach to classification and rules is a bottom-up classification approach. Finally, rule-based algorithms and decision tree algorithms differ from the other chosen algorithms in that they have an inherent ability to model interaction between the attributes of the categories [23]. We chose to investigate the conjunctive rules [68] algorithm, as it produces simple rules such as the example given above. We chose the C4.5 [50] algorithm as it is considered one of the best decision tree algorithms.

We evaluated two variations of the Naïve Bayes algorithm. The first is a multinomial variation and the second is an incrementally updatable variation. We evaluated the incrementally updated variation to investigate the use of an algorithm that is takes advantage of the flow of bug reports into the bug repository. We evaluated a multinomial variation of Naïve Bayes because it was used in previous work to automate bug assignment [19] and component assignment [23], and thus provided a lower bounds for finding a more appropriate algorithm. The algorithm is called *multinomial Naïve Bayes* because it uses a vector of integer features (i.e., word counts) to represent the reports.

We evaluated Support Vector Machines because it has been shown to be effective for text categorization [33] and previous work also indicated that it may be a good choice [23]. We chose to investigate the nearest-neighbour algorithm as it too had been used in previous work [23], and is representative of instance-based machine learning algorithms that do not precompute a model, but that use all of the reports for comparison. Finally, we investigated Expectation Maximization as it was suggested in previous work as a way to overcome the limitation of supervised machine learning algorithms that require reports be labeled [19].

Chapter 3

An Approach to Creating Triage Assisting Recommenders

Our approach to assisting bug report triage is to provide a variety of recommenders that guide the triager in making common development-oriented triage decisions. This chapter presents our triage-assisting recommender creation process, which we refer to as ML_{Triage} (i.e. a machine learning approach to creating triage-assisting recommenders). We begin by presenting an overview of the approach. Next, we show how we used one type of development-oriented decision recommender, a developer recommender, to tune the approach. We then present how to use ML_{Triage} to create recommenders for which product component to file the report against,[1] and which other project members may be interested in being informed of changes to the report. We conclude this chapter with an overview of Sibyl, a web service that uses recommenders created by our approach to provide development-oriented decision recommendations to triagers who use the Bugzilla issue tracking system.

3.1 Overview of ML_{Triage}

At a high-level, the creation of a recommender to assist with development-oriented triage decisions is straightforward. Figure 3.1 shows an overview of the process. First, reports from a project's issue tracking system are automatically selected. Next, specific pieces of data, called *features*, are collected from the selected reports; reports with similar features are grouped under a *label*. The label indicates the category or *class* to which the features belong. The extracted data and labels are then fed to a supervised machine learning algorithm and a recommender for a specific development-oriented decision is created. As in Chapter 1, we will refer to such a recommender as R_{DO}. When the recommender is asked to make a prediction for a new report, features are extracted from the new report and fed to the recommender, which provides a list of potential labels. Triage can be assisted by

[1]Without loss of generality, we use Bugzilla terminology throughout our description of our approach and the created recommenders.

applying this process to create different R_{DO}.

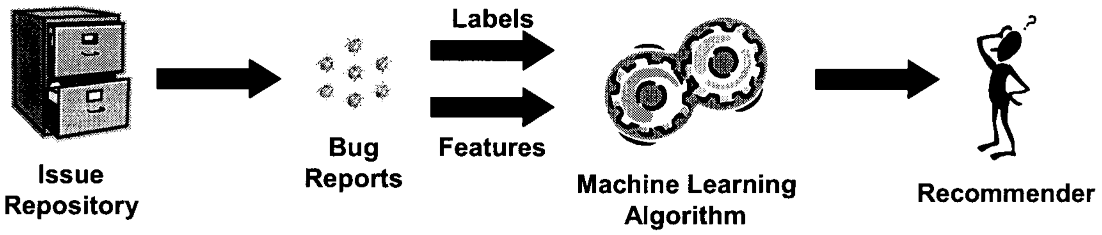


Figure 3.1: Overview of recommender creation.

Although the overall approach is straightforward, applying the approach in practice is complex as several inter-related decisions must be made to create the recommender. Specifically, six questions must be answered:

1. Which reports from the repository should be used to create the recommender?
2. How many reports should be used?
3. How should the reports be labeled?
4. What are the valid labels?
5. Which features from the reports should be used?
6. Which machine learning algorithm should be used?

Figure 3.2 depicts how the questions impact our overall approach.

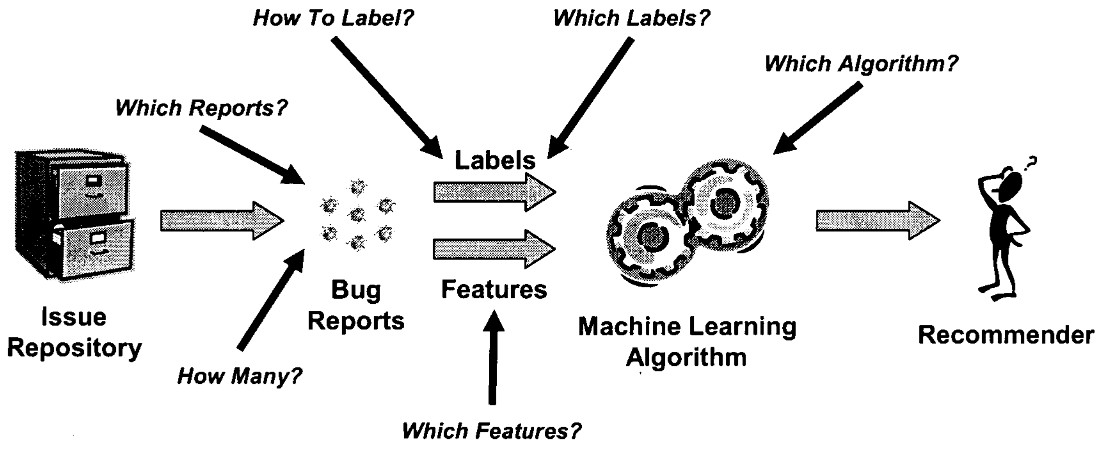


Figure 3.2: Questions to be answered in creating a recommender.

A similar process for answering these questions can be used to create different R_{DO}. It can be used to create a recommender that suggests which developer should be assigned the responsibility for resolving a particular bug (an assignment recommender or R_A). It can be used to create a recommender that suggests which project component the report should be filed against (a component recommender or R_C). It can be used to create a recommender that suggests which project members may have an interest in being informed of progress made towards the resolution of this bug (an interest recommender or R_I). This chapter describes in detail how to apply the overall process to create these three types of R_{DO} recommenders. We present other possible recommenders that can be created in Section 6.3.

We developed ML_{Triage} initially for creating a R_A. We then investigated how ML_{Triage} generalizes to other R_{DO}, namely R_C and R_I. This chapter provides a full analysis of answers to the questions for R_A, but limits the investigation of which reports, features, and algorithm to use for R_C and R_I.

As some of the questions require experimentation to answer, we tuned our approach using data from the Eclipse Platform[2] and Firefox[3] projects. We chose to use these two projects for tuning as they have large development communities, many project components, and have a large set of reports for training a recommender. These choices were then validated using five different projects (see Section 4.1).[4] We found that ML_{Triage} created recommenders with high precision (70% to 98%) for R_A, high recall (72% to 92%) for R_C, and R_I with moderate to high recalls (46% to 72%).

The remainder of this chapter describes how these three types of triage assisting recommenders are created using ML_{Triage}. The description of each recommender creation process is based upon how the six questions described above are answered.

3.2 A Developer Recommender

A developer recommender (R_A) provides the triager with suggestions of which project developers[5] should be given the responsibility of resolving the report based on historical information. For example, when a triager examines a newly submitted report, an R_A could suggest three developers who have the necessary expertise to fix the described problem.

In practice it is not possible to answer the questions for creating a R_{DO} recommender sequentially due to their inter-relation. We therefore do not describe the answers to the questions sequentially, but rather present the answers in a manner that minimizes the inter-dependencies.

[2]Eclipse provides an extensible development environment, including a Java IDE, and can be found at `www.eclipse.org` (verified 06/06/07).

[3]Firefox is a web browser and can be found at `www.mozilla.org/products/firefox` (verified 06/06/07).

[4]Without loss of generality, the projects that we investigate all used the Bugzilla issue tracking system. As explained in Section 2.2, the contents and life-cycle of reports are very similar across different issue tracking systems.

[5]A *project developer* is an individual that contributes to a project by resolving bugs or implementing features.

3.2.1 Which Reports?

Our approach requires selecting bug reports that provide information about how a project categorizes its reports. For a developer recommender this means selecting the reports that provide information about the problems each developer has resolved or the features that they have implemented. As discussed in Section 2.2.2, at any given moment, each report in an issue repository is at a different point in the report life-cycle. Some reports will be in a state, such as the UNCONFIRMED or NEW, that does not help in determining which developers have been known to resolve particular types of problems or implement certain features. As a result, we ignore these reports when creating a R_A, focusing instead on reports that have been either assigned to a developer or resolved.

As the approach for creating a R_{DO} uses a supervised machine learning algorithm (see Section 3.2.6), it is necessary that all reports in the training set have labels. We therefore further refine the data set by removing reports that cannot be labeled (see Section 3.2.4). The data set is also refined to remove reports that are labeled with the names of developers who our technique deem to not be actively contributing to the project at a sufficient level to warrant recommendation (see Section 3.2.5).

3.2.2 How Many Reports?

There are several ways to determine the quantity of reports from which to create a R_{DO}. One way is to use a fixed quantity, such as the five hundred most recent reports. However, this has two disadvantages. The first is that the appropriate number needs to be known in advance. The second is that different projects have different quantities of reports. Some projects may not have a pre-chosen fixed quantity, or if they do the reports may span a long time period, such as several years, increasing the likelihood of using of obsolete information.

Another way is to use a fixed percentage of the number of reports in the issue repository. However, this suffers from the same problems of using a fixed quantity: for projects with a large number of reports obsolete data may be used and for projects with a small number of reports, there may not be sufficient data to train the recommender.

Alternatively, the reports can be selected based on the time period over which they occurred, such as the reports that were resolved in the last six months. This technique for report selection has the advantage of providing some assurance of recency of information. We take the latter approach and use reports from the previous eight-month time period to create a R_A. This time frame was chosen empirically using data from the Eclipse and Firefox projects and appears to be appropriate for other projects (see Section 4.1.2). However, this time frame will depend on the specific project; some projects may not have been running for eight months yet still have a sufficient quantity of reports to create a R_{DO}.

Table 3.1: Date ranges for reports from the Eclipse and Firefox projects used for tuning R_A.

	Start Date	End Date
Eclipse	Oct 1, 2005	May 31, 2006
Firefox	Feb 1, 2006	Sept 30, 2006

3.2.3 Which Features?

Our approach requires an understanding of which 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 report. Reports with similar features can then be grouped.

As described in Section 2.2.1, each 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 a report. For resolved reports that have been marked as DUPLICATE, the text of both the report and the report it duplicates are used.

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 a standard text categorization approach [59] of first removing all stop words[6] and non-alphabetic tokens [6]. Although stemming[7] is traditionally used, we chose not to use stemming because earlier work [19] indicated that it had little effect. 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 (i.e., report) length, intra-document frequency and inter-document frequency [53].

If a term occurs once in a report, then it is likely to occur more times in the same report [53]. If the report description is long, this can skew the term distributions across all the reports. We therefore normalize a term's frequency by document length (t_r'), by dividing the term's frequency in the report (t_r) by the square root of the square of the term's frequency across all the reports (see Equation 3.1).

$$t_r' = \frac{t_r}{\sqrt{\sum_{i=1}^{|R|}(t_i)^2}} \tag{3.1}$$

The intra-document frequency (also known as *term frequency*) refers to how often a term occurs within the description of the report. We normalize the term frequency by taking the log of one plus the frequency of term i in report r (Equation 3.2). We do this so that terms with larger counts (low information content[8]) do not dominate terms with smaller counts (high information content). We add one to the term frequency so that the

[6]Stop words are functional words such as 'a', 'the', and 'of' which do not convey meaning.

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

[8]Information count is a measure of a feature's ability to distinguish between different classes.

transform becomes the identity transform for terms that occur zero or one time, since the word vector contains entries for all terms in the set of reports [53].

$$tf_{ir}^{'} = log(1 + tf_{ir}) \tag{3.2}$$

The inter-document frequency refers to the frequency by which terms occur across all the reports used to train R_A (i.e., the training set). We normalize by this frequency so as to discount terms that occur frequently across multiple documents. For example, terms that occur in the boiler plate of a report description such as those in the phrase "Steps to Reproduce" would not contribute information to distinguish reports that are labeled with the name of two different developers. To compute inter-document frequency for a term we take the frequency of term i in report r (f_{ir}) and multiply it by the log of the number of reports (n_r) over the number of reports containing term i (n_{ir}) (see Equation 3.3).

$$idf_i = f_{ir} * log\left(\frac{n_r}{n_{ir}}\right) \tag{3.3}$$

After this process, the feature vector for a report contains normalized frequency values for each of the words found in the summary and descriptions of the training reports.

3.2.4 How to Label the Reports?

To train the R_A being created, 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. At first glance, this step seems trivial as it seems obvious to use the value of the `assigned-to` field in the report. However, the problem is not this simple because projects tend to use the `status` and `assigned-to` fields of a report differently. For example, in both the Eclipse and Firefox projects, the value of the `assigned-to` field does not initially refer to a specific developer, but are first assigned to a default email address before they are assigned to an actual developer.[9] For reports with a trivial resolution, such as duplicate, or reports with a trivial fix, such as changing the access modifier of a method, the `assigned-to` field is often not 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 reports for the project. We took the latter approach for the Eclipse and Firefox projects resulting in a set of heuristics. Table 3.2 shows example heuristics for the Eclipse platform and Firefox projects. The full set of heuristics used for the various projects that were investigated is provided in Appendix A.

[9]Both Eclipse and Firefox use the Bugzilla issue tracking system and a user name in the Bugzilla system is an email address.

Table 3.2: Examples of heuristics used for labeling reports with developer names.

Eclipse	Firefox
If a report is resolved as FIXED, label it with whoever marked the report as resolved.	If a report is resolved as FIXED, label it with whoever submitted the last approved patch.
If a report is resolved as DUPLICATE, label it with whoever resolved the report of which this report is a duplicate.	
If the report was resolved as not FIXED by the person who filed the report and was not assigned to it, and no one responded to the bug, then the report cannot be labeled.	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. Label it as unclassifiable.

3.2.5 Which Labels Are Valid?

Having determined how to label the reports, we next decide which labels are valid labels, or which classes will be recommended. The set of valid labels is determined from the set of training reports.

However, before the labels can be determined, the set of training reports must be filtered. The first step in filtering is to remove reports that also occur in the set of reports used for testing. The removing of these reports is only relevant in the context of the evaluation of the recommender. For recommenders used by triagers, these reports would not be removed. The first column of Table 3.3 shows the number of reports for the Eclipse and Firefox training sets before any filtering. The next column shows the number of reports removed because they also occur in the set of reports used for testing. We also need to filter reports that cannot be classified by the labeling heuristics. The third column of Table 3.3 shows the number of unclassifiable reports. For the Eclipse project, only 1% of the reports in the data set are unclassifiable. In contrast, 39% of the Firefox reports are unclassifiable due to a large proportion (32%) of the training reports being marked as WORKSFORME, WONTFIX, INVALID, or the duplicate of a NEW bug report.[10]

All reports remaining after the filtering can be labeled. We now determine the set of valid labels by removing reports from the training set that are labeled with the name of developers who no longer work on the project and developers who have only fixed a small number of bugs. We remove the former because it is not useful to recommend a developer who is no longer available for assignment. We filter for the latter because we wish to recommend project members who have a demonstrated expertise with the project by making significant contributions to the project. We use the heuristics developed for the project (Section 3.2.4) to determine a developer's project contribution, which we call the developer's *activity level*. We use a threshold on the developer's activity level to determine the set of developers who warrant recommendation.

[10]These kind of reports are typically intercepted by a Firefox triager and it is unknown which developer would have been assigned responsibility for these reports.

Table 3.3: Size of data set, training set, and testing set used for process tuning.

	Reports	Removed			Training	Testing
		In Test Set	Unclassifiable	Filtering	Set	Set
Eclipse	7233	37	94 (1%)	746 (10%)	6356	152
Firefox	7596	15	2981 (39%)	1262 (17%)	3338	64

To determine an appropriate threshold, we evaluated the effect of choosing different activity thresholds on the precision and recall of a R_A using data from the Eclipse and Firefox projects. The recommenders used in this evaluation were created using the assigned and resolved reports from an eight-month time period (see Section 3.2.2), labeling heuristics (see Section 3.2.4), and the Support Vector Machines algorithm (see Section 3.2.6). We then varied the activity threshold value and determined the precision, recall and F-measure of the recommenders. Equations 3.4 through 3.6 show how precision, recall, and F-measure are calculated. Further details of how these values are determined is deferred to Section 4.1.1.

$$Precision = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ recommendations\ made} \tag{3.4}$$

$$Recall = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ possibly\ relevant\ values} \tag{3.5}$$

$$F = \frac{2 \times (precision \times recall)}{precision + recall} \tag{3.6}$$

Table 3.4 shows the effect of various developer activity profiles on recommendations for these two projects. The first column presents the activity threshold that we varied, ranging from no threshold to eighteen reports over the most recent three months. The next two columns shows the number of developers considered active for the Eclipse and Firefox projects when using the given threshold. The final two columns present the top one precision and recall of a R_A created using the given threshold. The first value in these columns is the precision and the second is the recall.[11] The table presents the results of R_A for one recommendation for brevity. We compare the effect of using no profile, including labeled reports from the most recent three months, and different contribution levels for the most recent three months. The time frame of three months was chosen based on an examination of activity profiles from the two projects spanning the period of a year and observing that many developers appeared to become inactive more than three months previous.

Table 3.4 shows that using the most recent three months removed 22 (21%) and 133 (37%) names from the Eclipse and Firefox data set respectively and that using a threshold value of an average of one resolution per developer per month for the three months also reduced the number of names an additional 68 (39%) and 121 (31%), for each project respectively. Given that the precision and recall values do not significantly change as these

[11]A detailed discussion of the ranges of precision and recall is deferred to Section 4.1.2 where our R_A evaluation procedure is explained.

Table 3.4: Precision, recall and F-measure when using developer profile filtering.

		# Dev.		Precision/Recall/F	
		Firefox	Eclipse	Firefox	Eclipse
No Profile		373	151	69/1/2	75/13/22
>=1 Fix in 3 mo.		240	129	66/1/2	75/13/22
Avg. Fixes Per Dev. Per Month Over 3 mo.	1	119	61	67/1/2	74/13/22
	2	68	53	66/1/2	74/12/22
	3	56	44	70/1/2	74/13/22
	4	40	42	70/1/2	74/13/22
	5	35	41	69/1/2	74/12/22
	6	33	38	69/1/2	74/12/22

names are pruned indicates that an appropriate set of developers to recommend is being found. This data confirms that using the most recent three months is appropriate for activity profiling.

Also, as there is no significant change in the performance of the recommender, any value between 1 and 18 reports in the most recent three months would be reasonable; we chose an average value of three resolutions over the recent three months as the threshold value. Using this threshold value filtered out 5% of the Eclipse reports and 17% of the Firefox reports from our data set (see the fifth column of Table 3.3). As we observed that the Eclipse heuristics labeled a number of reports with the default user names (i.e., the user names started with "platform"), we also removed reports with these labels, which removed an additional 5% of the Eclipse reports. The last two columns of Table 3.3 shows the size of the training and testing sets for the Eclipse and Firefox projects. The test set consists of the resolved reports from the month following the date range used for the training data set for which an implementation expertise set[12] can be determined

3.2.6 Which Algorithm?

There are a variety of machine learning algorithms that can be used to create a triage-assisting recommender (see Section 2.3.1). To determine an appropriate algorithm for R_A, we evaluated the effect of six different algorithms to create a R_A for the Eclipse and Firefox projects respectively. The algorithms we investigated were Naïve Bayes [34], Support Vector Machines [31], C4.5 [50], Expectation Maximization [22], conjunctive rules [68], and nearest neighbour [2].[13] We chose these algorithms as they cover the different categories of supervised machine learning algorithms [68], and Expectation Maximization provides an example of using unsupervised learning. We did not explore the use of reinforcement learning algorithms [64], because reinforcement learning focuses on learning as events happen,

[12]An *implementation expertise set* is a list of developers who we believe have the necessary expertise to resolve a report. See Section 4.1.2 for more details.

[13]The implementation of the algorithms was provided by the Weka machine learning library v. 3.4.7 (`www.cs.waikato.ac.nz/~ml/weka`, verified 17/07/07).

whereas we investigate learning from historical data.

We created R_A using the different machine learning algorithms and determined the precision and recall. Again, the description of how the precision and recall is determined is deferred to Section 4.1.2.[14]

The precision and recall of the recommenders created using six of the seven algorithms is presented in Table 3.5. As the seventh algorithm, nearest-neighbour, has an additional parameter — the number of neighbours to consider — Table 3.6 shows the precision and recall for this algorithm. For the nearest neighbour algorithm we explored the use of the nearest instance, and the five, ten, and twenty nearest instances. For a R_A, we are interested in a recommender that has high precision as we would prefer the recommender to produce a small list of developers with the right expertise as opposed to a recommender that produces a list containing all developers with the right expertise. From the tables we see that the Support Vector Machines and Naïve Bayes algorithms produced R_A that had the highest precision when making one recommendation. However, when making two or more recommendations, the Support Vector Machines algorithm generally provides a higher precision. We therefore chose Support Vector Machines as the algorithm for creating a R_A.

[14] The remaining answers for creating the R_A were the same as for determining the developer activity threshold (see Section 3.2.5)

Table 3.5: Precision, recall and F-measure of a R_A when using different machine learning algorithms.

Predictions	Naïve Bayes		SVM		C4.5		EM		Rules	
	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse	Firefox	Eclipse
	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)	(P/R/F)
1	76/1/2	72/12/21	70/1/2	75/13/22	67/1/2	57/10/17	35/0.4/1	11/1/2	66/1/2	24/2/4
2	63/2/4	52/17/26	65/2/4	60/20/30	64/2/4	46/17/25	38/1/2	12/2/3	41/1/2	27/9/14
3	61/3/6	48/23/31	60/3/6	51/24/33	65/3/6	35/19/25	41/1/2	11/3/5	41/2/4	24/11/15

Table 3.6: Precision, recall and F-measure of a R_A when using a nearest neighbour algorithm.

Predictions	k = 1		k = 5		k = 10		k = 20	
	Firefox (Prec./Rec.)	Eclipse (Prec./Rec.)	Firefox (Prec./Rec.)	Eclipse (Prec./Rec.)	Firefox (Prec./Rec.)	Eclipse (Prec./Rec.)	Firefox (Prec./Rec.)	Eclipse (Prec./Rec.)
1	56/1/2	24/4/7	58/1/2	16/3/5	58/1/2	28/5/8	53/1/2	37/5/9
2	64/2/4	18/6/9	59/1/2	14/4/6	57/2/4	17/6/9	51/1/2	36/10/16
3	59/3/6	13/7/9	47/2/4	11/5/7	55/2/4	18/9/12	50/2/4	32/14/19

Table 3.7: Answers to the six questions for ML_{Triage} when creating a component recommender.

Question	Answer
Which reports?	Assigned and resolved reports are used.
How many reports?	Reports are collected from an eight-month period.
Which features?	The summary and description are used for the features.
How to label?	**Reports are labeled by the value of the `component` field in the report.**
Which labels are valid?	**All component labels are considered valid.**
Which algorithm?	The Support Vector Machines algorithm is used to create the recommender.

3.3 A Component Recommender

In an issue tracking system, reports for large projects are often grouped by the functionality to which the report pertains. The Bugzilla issue tracking system refers to these groupings as *components*. A common occurrence with open bug repositories[15] is that reports are submitted under a default component, such as `General`. This results in the reports either needing to be regrouped during triage or causing a developer to be assigned who may not have the necessary expertise, possibly causing a delay in the resolution of the report. A solution to this problem is to provide the triager with a component recommendation so that the report can be correctly filed.

For example, when the triager examines an newly submitted report, a component recommender (R_C) could provide three recommendations of which project components the report should be filed against. If the project members are divided into teams based on the project components, then a R_C can be viewed as suggesting which project team to be assigned the report.

As previously mentioned, we found that most of the answers to the questions we determined from tuning R_A can be used to create an effective R_C. The two answers that differ is how report labels are determined and how valid labels are determined. Table 3.3 shows the six questions and their respective answers, with the differences in answers from creating R_A highlighted.

As a report will only belong to one project component, we are interested in knowing how well the recommender is at producing that correct recommendation. We therefore focus on recall over precision for a R_C. Using ML_{Triage} we created R_C for Eclipse and Firefox with recalls of 92% and 79% respectively for three recommendations. Further details are found in Section 4.1.3.

[15]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.

Table 3.8: Answers to the six questions for ML$_{Triage}$ when creating a interest recommender.

Question	Answer
Which reports?	Assigned and resolved reports are used.
How many reports?	Reports are collected from an three month period.
Which features?	The summary and description are used for the features.
How to label?	Reports are labeled by the names appearing in the cc: list of the report.
Which labels are valid?	Only names that occur on fifteen or more reports are considered valid.
Which algorithm?	The Support Vector Machines algorithm is used to create the recommender.

3.4 A Interest Recommender

It is common that a report will have a list of individuals who want to be notified when a change is made to a report. This list is maintained in the cc: list of a bug report. There are a variety of reasons that someone would be interested in changes to a report. If the report represents a problem, the individuals may be encountering the problem and want to know when the problem is fixed. If the individual is a developer for the project, they may be interested because they are working on a bug for which this bug is blocking their progress. Another reason is that perhaps a more senior developer would like to keep tabs on the work of a junior developer who he is mentoring. Finally, for some projects, such as Mozilla, where triagers do not assign reports but developers self assign, the list in the cc: field is used to inform developers of reports that they may want to fix.[16] An interest recommender (R_I) assists the triager in deciding which other project members should be aware of a report.

Again, we started creating a R_I with the answers determined during the tuning of R_A. We found that we needed to change how many reports were selected, how labels are determined and how valid labels are chosen. Table 3.4 shows these answers with the differences between the answers for a R_A and a R_I highlighted. In the rest of this section we provide more details about these three differences from a R_A.

As with R_C, for R_I we are interested in the recommender having a high recall. For the Eclipse and Firefox projects, ML$_{Triage}$ creates R_I with recalls of 49% and 46% respectively for seven recommendations. The discussion of why these recall values are acceptable is deferred until Section 4.1.4.

Unlike R_A and R_C recommenders where reports are gathered from a period of eight months, for a R_I we use a period of three months. The reason is that a single report can produce many training instances. Whereas reports are assigned to one developer and one component, the cc: list of a report can contain multiple names. For each of those names, a

[16]Personal communication with Mozilla developer, 01/03/05.

copy of the report is created and placed in the training set. If an eight-month time period is used, it can result in a very large training set (e.g., 16754 instances for our Firefox data set) that would likely tax a project's hardware resources, and the recommender would take a long time to train. Also, unlike the project expertise that a R_A captures, an individual's interest in a particular problem or set of problems is more likely to expire as their interest shifts to more pressing matters. Using a three-month period is more likely to capture the current interest of the individuals and not their expired interest.

As with a R_A, not all names appearing in the cc: list are appropriate for recommendation. Often the cc: list of a report contains the name of an individual who only appears on one or two other reports as they are directly affected by the problem. As there is very little data about the interest of such individuals, the recommender creation process cannot create a good model of their general project interest. We use a threshold on the number of times a name occurs to determine valid labels.

To determine an appropriate threshold for labeling, we performed an experiment whereby we varied the threshold value and fixed all other choices for the ML_{Triage} process used to create R_I. As opposed to a R_A where we were interested in recommending a small set of the right people, for a R_I we are interested in recommending all of the right people. In other words, for a R_A, we were interested in a high precision, and for a R_I, we are interested in a high recall. The precision and recall are computed by comparing the recommendations to the cc: list of the test reports. As with a R_A, the test reports assigned and resolved reports drawn from the month succeeding the period used for the training set.

Table 3.9 shows the results of using different thresholds for determining the set of names from which to recommend interest. As we are interested in judging if the recommender is suggesting all of the interested individuals, we report the precision and recall for seven recommendations. Again, the details of how the precision and recall were determined is deferred to Section 4.1.4.

We were unable to compute the precision and recall for the case where no threshold is used due to memory constraints.[17] Given the large number of labels (1000+) from which these recommenders suggest, it is doubtful that such a recommender would be of practical use because with this number of names, many of the names would have very few training instances and would not produce reliable interest models.

As with the developer activity threshold for a R_A, we again see stability in the average precision and recall for different interest threshold levels. We therefore choose to use the minimal threshold of fifteen for creating a R_I.

[17]We encountered OutOfMemory exceptions when using a 2 GB Java heap for the experiment.

Table 3.9: Precision, recall and F-measure when using cc: list filtering.

Threshold	Eclipse		Firefox	
	# Names	Recall/Precision/F	# Names	Recall/Precision/F
No Threshold	1146	-/-/-	1196	-/-/-
15 reports	58	49/13/21	108	46/15/23
25 reports	45	48/13/20	70	46/15/23
35 reports	37	47/12/19	57	46/15/23
45 reports	26	47/12/19	43	45/14/21

3.5 Sibyl : An Implementation of the Approach

To enable experimentation with human bug triagers, we integrated our triage-assisting bug recommendation approach into a web tool for the Bugzilla issue tracking system. We could have chosen another interface, but as the projects we examined all used Bugzilla and its web interface, this choice was practical for attempting to reach existing bug triagers. We call our tool Sibyl.[18] Sibyl consists of two parts: a back-end that provides the recommendations and a front-end that integrates the recommendations into the web page of the Bugzilla report. Both the front and back ends of Sibyl are implemented as Apache Tomcat[19] servlets.

3.5.1 Sibyl Back-end

The Sibyl back-end provides recommendations based on the report summary and description. Sibyl contains four types of R_{DO} recommenders:

1. R_A - which developer should be assigned the responsibility of resolving this report.
2. R_C - which project component should the report be filed against. If the project component has sub-components, Sibyl also recommends which sub-component the report should be filed against (R_S).[20]
3. R_I - which other developers on the project may be interested in being kept up to date on the progress of this report.

The recommenders are created as described in Sections 3.2 to 3.4. When the Sibyl front-end requests a kind of recommendation for a specific project, the appropriate recommender is loaded into memory[21] if need be (or reloaded if an updated recommender exists) and used. The recommenders are updated daily using a `cron` job. The `cron` job downloads and stores potential training reports from the previous twenty-four hours, adds these reports to the existing training data from the last several months, and then rebuilds

[18]A *sibyl* is a prophetess of the ancient world.

[19]Apache Tomcat is a servlet container that implements the Java Servlet and JavaServer technologies, and can be found at `tomcat.apache.org`, (verified 19/07/07).

[20]More information about the sub-component recommender can be found in Section 6.3.1

[21]Sibyl is implemented in Java and recommenders are stored as serialized objects on the server.

the project recommenders. To create the four recommenders for the Eclipse project takes approximately fifteen to thirty minutes depending on the number of new reports to be retrieved. For the Firefox project, it took an hour to an hour and a half to create a R_A and a R_C,[22] depending on how many new reports need to be retrieved.[23] Sibyl uses the implementation of Support Vector Machines in the Weka machine learning library v. 3.4.7 [24] to create the recommenders.

3.5.2 Sibyl Front-end

Figure 3.3 shows a screenshot of a report from the Eclipse project before Sibyl inserts the recommendations.[25] Figure 3.4 shows the same report after Sibyl inserts the recommendations with dotted boxes added to highlight the changes.

Figure 3.4 shows that Sibyl adds a drop down box for developer recommendations next to the "Reassign bug to" label (Figure 3.4-1). The triager still has the option of using the original text box to type in the name of the developer to whom to assign the report. Sibyl makes up to four developer recommendations.

For component recommendation, Sibyl augments the existing "Component" drop-down box (Figure 3.4-2) by inserting up to three component recommendations at the top of the list of values. A deliminator is inserted after the recommendations to distinguish between the recommendations and the list of components.

The sub-component recommendations are presented in a drop-down box that is inserted after the component drop-down box (Figure 3.4-3). As not all components have sub-components, this drop-down box is only inserted when the sub-component recommender can make recommendations. Sibyl makes up to three sub-component recommendations.

The final recommendation is the interest recommendations. These recommendations are presented in a multi-line text box (Figure 3.4-4). Sibyl makes up to seven interest recommendations.

We made the choices on how many recommendations to present based on our results of an analytical evaluation of the four recommenders. See Section 4.1 for the results of the analytical evaluation of R_A, R_C, and R_I. The analytical evaluation results for R_S are presented in Section 6.3.1.

Figure 3.5 provides an overview of how Sibyl provides recommendations to a triager. Sibyl works in the following manner:

[22] R_I and R_S were added to Sibyl during a field study with Eclipse triagers and after it was clear that no Firefox triagers would participate in the study. Therefore we do not have data on how long it would take to refresh the four recommenders for this project.

[23] So as not to overwhelm the repository server, the retrieval script has a ten second delay between subsequent report retrievals and can add a significant amount of time to the complete recommender creation process.

[24] Available at `www.cs.waikato.ac.nz/~ml/weka`, verified 17/07/07.

[25] Both Figure 3.3 and Figure 3.4 show condensed versions of the actual Bugzilla interface.

Figure removed for copyright reasons.
Original source found at
https://bugs.eclipse.org/bugs/show_bug.cgi?id=193787

Figure 3.3: The Bugzilla interface before Sibyl inserts recommendations.

Figure removed for copyright reasons.
Original source no longer available.

Figure 3.4: The Bugzilla interface after Sibyl has inserted the developer, component, sub-component, and interest recommendations.

1. A request to view a report from the project's Bugzilla repository is intercepted in the triager's web browser by a Javascript script. The request is then redirected to the Sibyl server.
2. The Sibyl front-end forwards the request to the project's Bugzilla repository.
3. The Sibyl front-end receives the report page from the project's Bugzilla server. It then extracts the summary and description from the report and gets the developer, component, sub-component, and interest recommendations from the Sibyl back-end. Next, the Sibyl front-end inserts the recommendations into the report page.
4. The Sibyl front-end returns the augmented report page to the triager's browser.

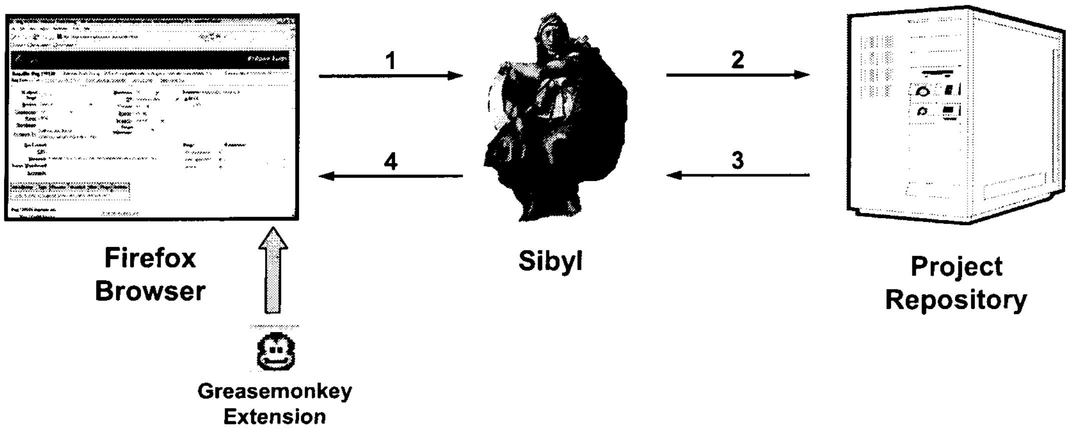


Figure 3.5: Overview of how Sibyl works.

To redirect the triager's report viewing request, we use Greasemonkey. Greasemonkey,[26] is a Firefox extension that allows the running of user-created Javascript scripts in the browser. As part of the user configuration of Sibyl, the triager installs a generated Javascript that detects when the triager is accessing a report from a known Bugzilla repository, and redirects the request to the Sibyl server. The triager can deactivate the script or the Greasemonkey extension at any time if they did not want to use Sibyl.

3.6 Summary

This chapter presented our approach to creating recommenders to assist the triager with development-oriented decisions (R_{DO}). To create such a recommender requires answering six inter-related questions. A substantial number of answers to these questions are the same for the different kinds of R_{DO}. The answers to the three questions of which reports to use, what features to use, and what algorithm to use remain the same regardless of the kind of R_{DO}. The variation in the recommenders is in how many reports to use, how the reports are labeled, and which are the valid labels. We presented the details of how we used the creation of a R_A to answer and tune the process, and used these results to apply the ML_{Triage} to creating recommenders for the project component the report should be filed against and for which other project members may have an interest in being made aware of the report. We found that this approach created R_A with precisions of 70% and 75% when making one recommendation, for Eclipse and Firefox respectively. We found that using ML_{Triage} to create a R_C created recommenders for the Eclipse and Firefox projects with recalls of 92% and 79% respectively for three recommendations. Finally, we created R_I for Eclipse and Firefox with recalls of 49% and 46% respectively.

Finally, we presented a description of the implementation of a web service that made triage-assisting recommendations using recommenders created by ML_{Triage}.

[26]`http://www.greasespot.net/` verified 21/05/07.

Chapter 4

Evaluation

Bug triage is a human-intensive process that consumes resources that could be better utilized elsewhere in the project. In this dissertation we have introduced a machine learning-based triage-assisting recommender creation approach (ML_{Triage}) that we hypothesize can reduce human involvement for development-oriented decisions in the bug triage process. We formulate three research questions to investigate this hypothesis.

R1: **Does $\mathbf{ML}_{Triage}$ create recommenders that make accurate recommendations?** If ML_{Triage} creates recommenders that make accurate recommendations, then a triager need not examine the report as deeply as they would without the recommendations. Using such recommenders changes the triager's role from making decisions relying on their own knowledge, experience, and intuition or that which they can gain from existing tools to confirming decisions made by the recommender.

R2: **Do recommenders created using $\mathbf{ML}_{Triage}$ reduce the time it takes to triage a report?** If ML_{Triage} recommenders reduce the time taken to triage reports, then some of the human resources consumed by the triage process can be directed elsewhere in the project. This redirection of resources towards development is especially important for projects where developers also have the role of triager.

R3: **Do the recommenders created using $\mathbf{ML}_{Triage}$ help triagers to make better decisions?** In some projects, triage is delegated to those who do not yet have the project experience to consistently make correct development-oriented decisions. When errors are made, more human involvement is necessary as one or more experts must assist the triager to correct the mistake. For example, if a triager makes an incorrect developer assignment, the incorrectly assigned developer must inform the triager that a mistake was made and may further have to explain the reason and help the triager find the correct developer. In either case, there has been more human involvement in the triage process than if a better decision had been made.

To investigate R1, we conducted an analytical evaluation of three development-oriented decision recommenders: a developer assignment recommender (R_A), a component

Table 4.1: Overviews of the five projects used for analytic evaluation.

Project	Triage Process	Domain	Contributors
Eclipse	Developer-driven	Programming Tool	151
Firefox	Volunteer-based	Web Browser	343
gcc	Developer-driven	Compiler	81
Mylyn	Developer-driven	Programming Tool	13
Bugzilla	Volunteer-based	Issue Tracking	43

recommender (R_C), and an interest recommender (R_I). Our analytical evaluation consisted of a series of laboratory experiments where the recommendations were compared to data extracted from reports. We investigated R2 and R3 through a field study, in which four human triagers from the Eclipse project used Sibyl over a four-month period (see Section 4.2).

4.1 An Analytic Evaluation of the Recommenders

To validate our belief that ML_{Triage} can produce good recommenders, we analytically evaluated R_A, R_C, and R_I (created as described in Chapter 3) through laboratory experiments using data from five projects: the Eclipse Platform project, the Firefox project, the gcc compiler project,[1] the Mylyn project,[2] and the Bugzilla project. These projects were chosen as they span a range of triage processes and domains, small to large numbers of contributors, and low (< 10 per day) to high (> 20 per day) bug submission frequencies. Table 4.1 shows the different dimensions for the projects. The second column categorizes the projects based on their triage process: developer-driven or volunteer-based. The next column describes the application domain of the different projects. The fourth column gives an estimate[3] of the number of contributors[4] for each project based on our eight-month data sets. In other words, the number of contributors for a project is the number of distinct developer labels for all the reports in our data set. Table 4.2 shows the minimum, maximum, and average number of reports submitted daily to each project over the eight-months period covered by our data sets (see Table 4.3). As mentioned in Chapter 3, we used the Eclipse and Firefox projects for tuning ML_{Triage}. We report the results for all five projects in this section, including the tuning results for ease of reference.

This section begins with an overview of our evaluation methodology. The remainder of this section presents the results of our analytic evaluation of the three kinds of recommenders.

[1]Gcc is a collection of compilers and can be found at `www.gnu.org/software/gcc/gcc.html` (verified 06/06/07).

[2]Mylyn is a task-focused UI plug-in for Eclipse and can be found at `www.eclipse.org/mylyn` (verified 06/06/07).

[3]As these values are based on our project heuristics, we can only assert that this is an estimate.

[4]Recall that we define contributor to mean an individual that contributes to source code of the project.

Table 4.2: Report submission statistics for the five projects.

Project	Report Submitted / Day		
	Min.	Max.	Avg.
Eclipse	0	129	29
Firefox	12	103	36
gcc	0	23	9
Mylyn	0	15	4
Bugzilla	0	18	3

Table 4.3: Date ranges for data used for the analytic evaluation.

	Start Date	End Date
Eclipse	Oct 1, 2005	May 31, 2006
Firefox	Feb 1, 2006	Sept 30, 2006
gcc	Apr 1, 2006	Nov 30, 2006
Mylyn	Feb 1, 2006	Sept 30, 2006
Bugzilla	Feb 1, 2006	Sept 30, 2006

4.1.1 Overview of Analytic Evaluation Procedure

The general methodology we used to analytically evaluate the recommenders was to train the recommender on the data from several months and then use the resolved reports from the following month as the testing set. We chose this approach to ensure a realistic evaluation; in practice, one would be using a recommender that was trained on the most recent information.

As is typical in machine learning, we evaluate the performance of the recommenders created using ML$_{Triage}$ by the measures of precision and recall. Precision measures how often the approach makes a relevant recommendation for a report (Equation 4.1). Recall measures how many of the recommendations that are relevant are actually recommended (Equation 4.2). For example, for a R_A the precision measures the percentage of recommendations of a developer with relevant experience to fix the problem and recall measures how many of the developers who had the appropriate expertise were recommended. As precision and recall provide complementary measures, F-measure is used to express the harmonic mean of the precision and recall (Equation 4.3).

The key piece of information in computing both precision and recall is the set of appropriate recommendations. As this information is specific to the kind of recommender, we defer explaining how we determine this set for each recommender to the appropriate sections.

$$Precision = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ recommendations\ made} \tag{4.1}$$

$$Recall = \frac{\#\ of\ appropriate\ recommendations}{\#\ of\ possibly\ relevant\ values} \tag{4.2}$$

Table 4.4: Training and testing set sizes for evaluating developer recommenders.

Project	Developers	Training Reports	Testing Reports
Eclipse	31	6356	152
Firefox	31	3338	64
gcc	31	2521	70
Mylyn	6	683	50
Bugzilla	11	799	52

$$F = \frac{2 \times (precision \times recall)}{precision + recall} \tag{4.3}$$

4.1.2 Developer Recommender Evaluation

We used the data from three other projects, gcc, Mylyn, and Bugzilla, to evaluate the R_A; the data for Eclipse and Firefox was used to tune the R_A process and is provided for convenient reference. Table 4.4 shows the number of active developers, the number of training reports, and the number of testing reports used in the evaluation. This data shows that the gcc project has a similar number of active developers as the Eclipse and Firefox projects (see Table 3.3) and that the Mylyn and Bugzilla have few active developers.

Determining the Implementation Expertise Set

To evaluate a R_A, we need to know for each report in the test set which developers on the project might have the implementation expertise to resolve the report. Implementation expertise refers to expertise in the code base of the software product and is needed for determining the precision and recall of a R_A. The easiest way to determine the set of developers with implementation expertise for a report is to ask an expert in the project. We did not have access to such experts so we developed heuristics for each project based on information in the source revision repositories.

Source revision repositories log information about changes made to the source code each time a developer submits code to a source repository (also known as checking-in or committing). The submission log entry includes various pieces of meta-information, such as the file that was checked in, its revision number, the check-in date and time, and the check-in comment made by the developer performing the commit. The source repository software can produce all of the log entries for a specific file. We use these source repository check-in logs to provide information about the implementation expertise of the project developers, that is which developers have accumulated expertise in which parts of the project's code base. We chose to use source repository information so as to improve the construct validity of our R_A evaluation. Also, this technique has also been used by others for the same purpose [43, 44, 47].

We refer to the set of developers who have implementation expertise for a particular report as the report's *implementation expertise set.* Creating an implementation expertise set for a report involves three steps.

The first step is to establish the linkage between the report and the source repository [27, 28, 61]. For projects that use tools that automate this linkage, this step is trivial. Unfortunately, this was not the case for the projects that we examined.[5] However, each of the projects does use the same linkage convention of putting the report id in the check-in comments. To establish a report-source code link we search the source repository logs for the report id number and collect the names of the corresponding files. The resulting list of source files forms the *change set* for the report [70].

The next step is to determine the containing module for each source file in the change set. The *containing module* may refer to the file itself or some higher abstraction, such as its package in the case of software implemented in Java. This step recognizes that it may be unrealistic to examine just the source files of a particular fix when determining expertise. Developers who work with associated files, such as those in the same package, also have some level of implementation expertise that may be relevant and this expertise should not be discounted. For the Java projects (Eclipse and Mylyn), we used the package as the containing module. For the Mozilla projects (Firefox and Bugzilla), we used the sub-component.[6] For gcc, we used the directory in which the file appears.

The last step is to compile a set of developers who had previously committed changes to each module. This set is constructed by analyzing the revision history of each file in the module and using the Line 10 Rule [42, 43, 44, 47]. The Line 10 Rule is a heuristic in which line 10 of the source repository check-in log for a particular file, the line containing the user name of the person who performed the commit, is used to determine who has expertise for that source file. For each file in the containing module, we gather the names of all users who have committed a change for the file to form the implementation expertise set for that file. We then combine the implementation expertise sets for all the files in the containing module to create the implementation expertise set for the bug report.

Using this technique we determine for each report, the list of developers who have the implementation expertise to resolve the report. Unfortunately, we cannot directly use this list to calculate the precision and recall of the R_A. As the issue repository and source repositories are disjoint systems, a developer is often represented in each system by different user names [7]. As a result, we have to create a mapping between the user names of the two systems. For each project, we created a mapping from the user names in the issue tracking system to those found in the source repository.

For the Eclipse, gcc, and Mylyn projects this mapping is straightforward, as developers who make a fix usually have commit access to the repository. The mapping for

[5]Others have also observed that mapping bug reports to the source changes can be difficult. For example, Williams and Hollingsworth were only able to link 24% of reports to changes in the source repository for the Apache web server [66].

[6]For some projects, the product components are further subdivided into sub-components.

Firefox and Bugzilla is more challenging because these projects use a process in which fixes are reviewed and contributors[7] must request that their changes be checked in by a committer.[8] As a result, the person who checks in a fix is rarely the person that made the fix. In constructing a user names mapping for such a project, we assumed that the person who marked a report as resolved is generally also the person who checked in the fix. We then mapped the user name of the individual that marked the report as resolved, to a list of user names on whose behalf they may have committed changes to the source repository.

Using the technique of mapping of bug reports to names from the source repository has an important consequence: it overestimates the set of developers with implementation expertise, especially for projects such as Firefox and Bugzilla where an additional mapping of resolvers to fixers is needed.[9] As the denominator of the recall formula is the size of this set, the computed recall value will be lower than the true value. However, as this overestimation of the developers makes it less likely to miss a developer with the relevant expertise, this larger error in the computed recall is compensated by a lower error in the computed precision. In other words, as we are using an approximation of the correct set of developers who have the needed expertise for a report, there is a measure of error in our calculation of precision and recall for a R_A. However, the measure of error is less for the precision calculation than for the recall calculation. This smaller amount of error is important for the construct validity of our R_A results, as we favour precision over recall.

There are other techniques that may be used to determine the implementation expertise set for a bug report. For example, the bug network [58] that a report is in can be used for creating such a set. To determine which technique was most appropriate, we evaluated how well the bug network technique and the source repository technique are at approximating the correct implementation expertise set [5]. We found that the source repository technique was better suited for the evaluation of a R_A. [5]

Results

Table 4.5 shows for the five projects the size of the pool from which recommendations are drawn and the average size of the implementation expertise lists used for the evaluation. Table 4.6 shows the results of applying ML_{Triage} to create a R_A for the gcc, Mylyn and Bugzilla projects, as well as the results from tuning with Eclipse and Firefox. Figure 4.1 shows a graph of the precision, recall and F-measure for the top recommendation for each of the five projects. From the data, we see that the process creates recommenders with high

[7]A committer is a project member that has the privileges necessary to commit changes to the source repository.

[8]Although this situation may also happen with the Eclipse, gcc, and Mylyn projects, we observed that it was the exception and not the rule. In other words, most changes seemed to have been made by committers.

[9]Although this overestimation could lead to an artificial inflation of the precision values, other work [5] suggests that any artificial inflation will not be significant.

Table 4.5: Size of recommendation pool and average implementation expertise list size.

	Eclipse	Firefox	gcc	Mylyn	Bugzilla
Known Developers	44	56	31	6	11
Avg. Implementation Expertise List Size	8	80	43	3	32

Table 4.6: Precision, recall and F-measure for the five developer recommenders.

	Eclipse (P/R/F)	Firefox (P/R/F)	gcc (P/R/F)	Mylyn (P/R/F)	Bugzilla (P/R/F)
1	75/13/22	70/1/2	84/3/6	98/30/46	98/5/10
2	60/20/30	65/2/4	82/6/11	93/55/69	98/11/20
3	51/24/33	60/3/6	76/10/18	82/72/77	92/14/24

precision for all these projects. These precision values indicate that for a reasonable sized set of recommendations, such as two or three, the set will often contain an appropriate developer. The precision for the gcc, Mylyn, and Bugzilla projects is better than that for the Eclipse and Firefox projects likely because of the smaller development teams for these three projects. As opposed to the Eclipse and Firefox projects, where the R_A makes recommendations from forty to fifty developers, for these projects the recommender chooses between six (Mylyn), eleven (Bugzilla), or thirty-one developers (gcc). With a smaller number of developers to recommend from, it is more likely that the recommender will make a correct recommendation. This is supported by the lower precision for gcc, relative to the other two projects, which has roughly three to five times as many developers from which to make a recommendation.

Compared to the other projects, the R_A for Mylyn has a very high recall. We believe this is a result of our evaluation methodology. Of all the projects, Mylyn has the smallest pool of developers who might have expertise for a particular report. Therefore, the overestimation of the developers with implementation expertise for a particular report will be less and the error in the recall value will be less.

4.1.3 Component Recommender Evaluation

As we described earlier in Section 3.3, reports in a bug repository are commonly grouped around the functionality that they affect and that on initial submission, reports are often misfiled with respect to the component[10]. This section presents the results of using ML_{Triage} to create a component recommender (R_C)for the five projects.

As a report will only belong to one project component, we are interested in knowing how well the recommender is at producing that correct recommendation. As there is only one correct answer, computing the precision is not enlightening; the precision for one recommendation will be same as the recall, and the best precision that could be achieved

[10]See `http://wiki.mozilla.org/MozillaQualityAssurance:Triage`, verified 01/08/07.

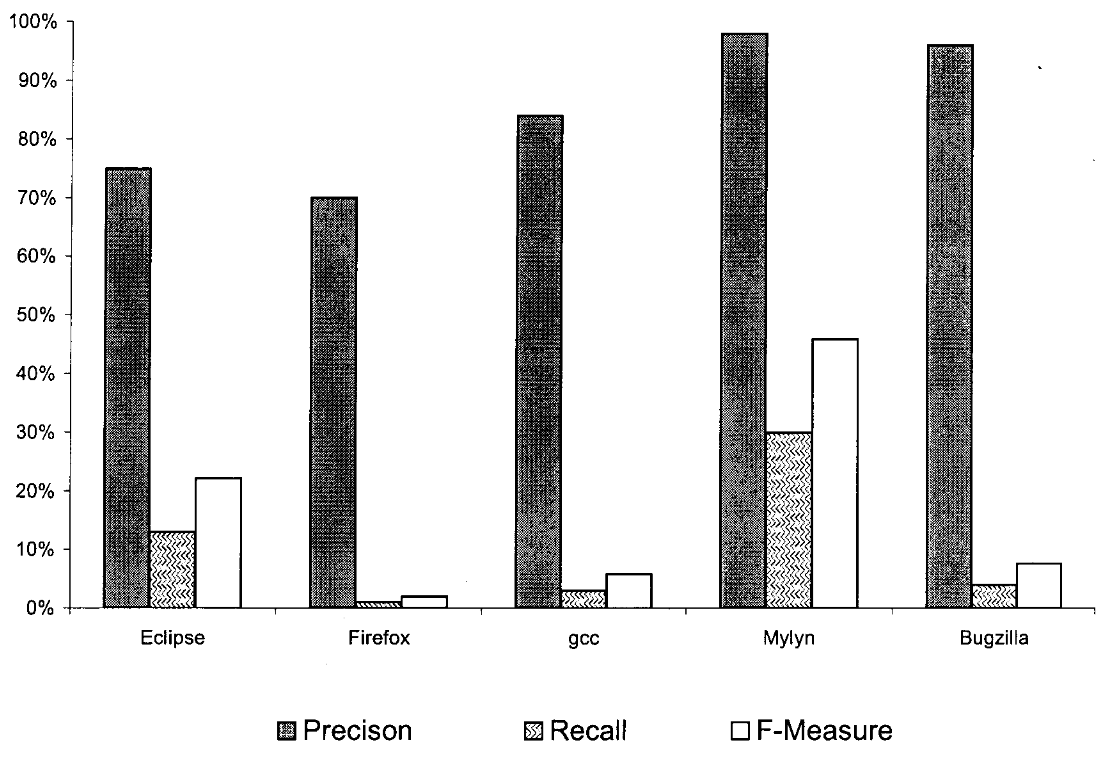


Figure 4.1: Graph of the five developer recommenders for the top recommendation.

for the two and three recommendations would be 50% and 33% respectively. We therefore focus on the recall of the recommender and do not report precision.

The number of components and size of the training and tests for the evaluation of the component recommenders is shown in Table 4.7. To compute the component recommender's recall, we used the value of the testing report's component field as the relevant component for the report.

Table 4.8 shows the results for using ML_{Triage} to create a R_C for each of five different projects. We see from the table that if we use the top recommendation (the row labeled "1" in Table 4.8), the recommender correctly identifies the component 50% to 66% of time. If three component recommendations are made then the correct component is identified 75% of the time or better.

The R_C works better for some projects, such as Eclipse, than for others. We believe there are two likely reasons. The first reason is that the Eclipse project has the highest number of training reports per component (376 reports) of the projects. Compare this to the Bugzilla project which has a similar number of components but one-sixth of the number of training reports as Eclipse and has the lowest recall. In other words, Eclipse has more information about each component.

Alternatively, this may be a result of who is filing the reports. If most reports are filed by project members, then the report descriptions will likely have similar terminology and the recommender will have an easier time determining the component boundaries. Table 4.9

Table 4.7: Number of components, and size of training and testing sets for component recommender evaluation.

	Eclipse	Firefox	gcc	Mylyn	Bugzilla
Components	18	34	33	11	16
Training Reports	6759	6302	2030	751	1012
Testing Reports	1489	969	275	94	158

Table 4.8: Recall for the five component recommenders.

Predictions	Eclipse (%)	Firefox (%)	gcc (%)	Mylyn (%)	Bugzilla (%)
1	66	57	55	53	45
2	85	72	68	66	62
3	92	79	76	83	74

shows the percentage of reports from our data set that were submitted by active developers. We found that 45% of the reports in the training set for the Eclipse recommender were submitted by active developers, as compared to 18% for the Firefox project, lending some support to this idea. However, both the Mylyn and Bugzilla projects have a similar number of components and training reports per component and yet the Mylyn project has a higher recall even though fewer (39% compared to 45%) of the reports are submitted by active developers. It therefore may be that the high recall for Eclipse is a result of a combination of both the high number of training reports per component and the general background of the reporters.

4.1.4 Interest Recommender Evaluation

As described in Section 3.4, a report often contain a list of individuals who are interested in being notified when a change is made to the report. This section presents the results for using ML_{Triage} to create a recommender for who might be interested in a particular report, R_I (Section 3.4).

Table 4.10 shows the number of names and the size of training and tests sets for the five projects. As discussed in Section 3.4, the training reports are taken from the most recent three months of our training data set (see Table 4.3) and the test reports are drawn from following month. We use the cc: list of each test report as the set of correct values for the report.

Table 4.9: Percentage of training reports submitted by active developers.

	Submitted By Active Developer
Eclipse	45%
Firefox	18%
gcc	24%
Mylyn	39%
Bugzilla	45%

Table 4.10: Number of names, and training and testing set sizes for interest recommendation.

	Names	Training Reports	Test Reports
Eclipse	58	3256	860
Firefox	108	7901	790
gcc	21	1811	275
Mylyn	5	172	31
Bugzilla	17	697	108

Table 4.11: Precision, recall and F-measure for the five interest recommenders.

Predictions	Eclipse (P/R/F)	Firefox (P/R/F)	gcc (P/R/F)	Mylyn (P/R/F)	Bugzilla (P/R/F)
1	22/11/15	29/13/18	100/59/74	29/21/24	42/24/31
2	21/22/21	26/21/23	61/65/63	18/28/22	32/34/33
3	19/31/24	23/29/26	44/69/53	16/38/23	30/46/36
4	17/37/23	19/33/22	35/69/46	15/47/23	26/53/35
5	16/43/23	18/39/25	29/71/41	14/54/22	24/57/34
6	14/47/22	16/43/23	24/72/36	-/-	23/62/34
7	13/49/21	15/46/23	21/72/33	-/-	21/65/32

Table 4.11 shows the results for the R_I of the five projects for up to seven recommendations.[11] We chose to investigate the presentation of seven R_I recommendations, as we want to determine how many recommendations need to be presented for the recommender to have a good recall. As with R_C, we favour recall over precision for an R_I (Section 3.4).

The data in Table 4.11 shows that the R_I recommenders achieve recall levels of between 46% to 72% for seven recommendations (or five in the case of Mylyn). In other words, a R_I will correctly recommend roughly 50% to 75% of the names that appear on the cc: list of the report. Considering that a cc: list will often contain names of individuals who appear on the cc: list of very few reports (see Table 3.9), it seems unreasonable to expect a high recall value from an cc: list recommender in general. We therefore believe that having a recall between 46% to 72% is a good range to expect from such a recommender.

4.1.5 Threats to Validity

A threat to the construct validity of our analytic evaluation of the R_A is our technique for determining implementation expertise. As was stated previously, the technique overestimates the number of developers who have implementation expertise for a particular reports. Although this improves the accuracy of our precision measurement, it does so at the expense of hindering our ability to accurately determine the recall of the R_A. As we were more interested in the precision over the recall of this type of recommender, we do not feel that this is a problem.

A similar threat to the internal validity of the analytical evaluation of the R_A is the

[11]The Mylyn R_I only had five names which met the threshold criteria of 15 reports.

mapping of user names between the bug and source repositories. Although we made every attempt to make correct mappings, it is possible that some of the mappings were in error.

4.1.6 Summary of the Analytical Evaluation

We conducted an analytic evaluation to answer our first research question: "Does ML_{Triage} create recommenders that make accurate recommendations?" For a R_A we were looking for a high precision and for a R_C and a R_I we were looking for a high recall.

We found that ML_{Triage} created a R_A with reasonably high precision, ranging from 70% for Firefox to 98% for Bugzilla and Mylyn when making one recommendation. We found that using ML_{Triage} to create a R_C resulted in recommenders that correctly identified the project component 72% to 92% of the time when making three recommendations. Finally, for using ML_{Triage} to create a R_I, we found that it produced a R_I with recalls between between 46% to 72%. Given that there will commonly be individuals that have an interest in a report, but are unknown to the recommender, we felt that this recall range indicated that ML_{Triage} does produce a R_I that are accurate, even though the recall was not generally high.

4.2 A Field Study of the Recommenders

Having established analytically that ML_{Triage} produces recommenders that make good recommendations, we conducted a field study to answer four questions. The first two questions were R2 and R3:

R2: **Do recommenders created using ML_{Triage} reduce the time it takes to triage a report?**

R3: **Do the recommenders created using ML_{Triage} help triagers to make better decisions?**

The other two questions were about how well the recommenders worked in practice (R4) and about one of the Sibyl design choices (R5).

R4: **What is the empirical performance of recommenders created using ML_{Triage}?** We wanted to compare whether the results from the analytic evaluation correspond to the results seen in practice.

R5: **What is an appropriate number of recommendations to present to the triager?** We wanted to determine what the human triagers thought the appropriate number of recommendations would be for R_A, R_C, R_S, and R_I.

4.2.1 Study Subjects

We recruited triagers from four different open-source projects as subjects for our study. The four projects we recruited from were Eclipse, Firefox, Mylyn, and Evolution[12]. The subjects were asked to use Sibyl as part of their normal triage activities for a period of three months with their use of the tool logged. Subjects were recruited via project news groups, and direct contact. For participating in the study, subjects were given a $20 gift certificate for an online bookstore.

As part of the user configuration of Sibyl, the participants were asked to answer a background questionnaire. The questions asked of the participants are shown in Figure 4.2.

1. What is your job function? [Application Developer, QA/Testing, Program Director, CIO/CTO, VP Development System Integrator, Application Architect, Project Manager, Student, Faculty, Business, Analyst, Database Administrator, Other]
2. How many years of programming experience do you have?
3. Are you a developer for the project? [Yes, No]
4. How long have you worked with this project?
5. How many years of triage experience do you have?
6. How many hours a week do you spend triaging?
7. How many triagers are there for your component or project?
8. When assigning reports, what criteria do you use? Why is a report assigned to a particular person?

Figure 4.2: Questions for the background questionnaire.

Only triagers from the Eclipse Platform project agreed to participate in the field study. Four triagers from the User Interface (UI) team of Eclipse used Sibyl successively for four months as part of their triage duties. Although the subjects were asked to use Sibyl for three months, this proved impractical, as triage duty is rotated among the six project members, with each member taking a six-week turn.

The four subjects reported that they spent anywhere from one to six hours doing triage during their turn and that their triage experience for the project ranged from just over a year to off-and-on for five years. Table 4.12 shows the reported project experience, triage experience, and weekly time spent triaging reported by the four subjects.

[12]Evolution is a personal information management product from the Gnome project and is available at `www.gnome.org/projects/evolution`, verified 23/08/07.

Table 4.12: Triage experience as reported by triagers.

	Project Experience	Triage Experience	Weekly Triage Time
Triager A	3 years	1 year (18 weeks - 3 milestones)	6 hours
Triager B	3 years	2.5 years	2-3 hours
Triager C	5 years	Off and on for 5 years	1 hour
Triager D	1 year	1 year (18 weeks - 3 milestones)	5 hours

4.2.2 Study Methods

This section provides an overview of the methods we used during the field study. We begin by describing the recommenders used by the subjects. Next, we discuss how we monitored their activity. We then present information about the questionnaires we used to collect various background and qualitative data. Finally, we present overviews of the quantitative and qualitative techniques we used for analyzing the collected data.

Study Recommenders

The version of Sibyl used by the subjects provided four types of recommendations: developer recommendations (R_A), component recommendations (R_C), sub-component recommendations (R_S)[13], and cc: list recommendations (R_I).

The R_A used by the subjects was a component-based recommender, which makes developer recommendations based on the component against which the report is filed (see Section 6.2.1). The R_C and R_I used by Sibyl were created using the same approach as described in Chapter 3. The sub-component recommender (R_S), was created similar to R_C but sub-components, unlike component designations, are free-form text fields rather than a configured field.

The only other difference between the recommenders used for the analytical evaluation and those used in the field study was the date range from which the reports were collected. For the field study, the eight months of reports were collected starting from the previous day of the recommenders' use. For example, if the triager was using the recommenders through Sibyl on January 1, then the recommenders were trained with project data from to May 1 to December 31 of the previous year. A daily system job was used to gather the new training data and save it to disk.

As active data was being collected, it occurs that an assigned report collected one day could be collected as a resolved report on a subsequent day. To use the report with the most recent information, reports were gathered into the training set in reverse chronological order before training the recommender. In other words, the most recent version of the report was used to train the recommenders.

[13]The sub-component recommender was only applicable to the Eclipse project and is discussed in Section 6.3.1

1. The number of recommendations given were: [Too many, Reasonable, Too few]
2. Were multiple recommendations appropriate? [Yes, No]
    - If Yes, was choosing between them easy or difficult? [Very easy, Reasonably easy, Reasonably difficult, Very difficult]
    - Please explain why.

Figure 4.3: Questions for the decision questionnaire.

1. How useful are the assignment recommendations? [Very useful, Reasonably useful, Not very useful]
2. Does it make it faster to do assignments? [Yes, No]
3. Does it make you consider someone you might not have previously considered? [Yes, No]
4. Are there any systematic differences between your expectations and what was recommended? [Yes, No]
5. If you answered Yes to Question #4, please elaborate.
6. Do you have any comments about using the recommender?

Figure 4.4: Questions for the usage questionnaire.

To provide a base-line for how long it takes a triager to triage a report, the Sibyl back-end designated every fifth recommendation request as a control case and made no recommendations. Users were informed of this at the start of the study so as not to be surprised when no recommendations were made.

Questionnaires

In addition to the quantitative data collected through the use of Sibyl, the subjects were asked to complete questionnaires. There were two questionnaires.

The first questionnaire was presented periodically during the subject's use of Sibyl and included questions about the number of recommendations presented and if multiple recommendations were appropriate (see Figure 4.3). These questions were asked for each of R_A, R_C, and R_S. We refer to this questionnaire as the *decision questionnaire.* This questionnaire was presented to the subject in the web browser after every tenth recorded report change.

The second questionnaire was presented in the web browser when the subjects submitted their activity logs for analysis and asked about the subject's impressions of using the recommenders. The questions for R_A are shown in Figure 4.4. The questions for R_C, R_S, and R_I were similar. We refer to this questionnaire as the *usage questionnaire.*

1. Time stamp of when a triager viewed a report.
2. Time stamp of change request and the changes to a report submitted by a triager.
3. The recommendations made by Sibyl for the different kinds of recommenders.
4. The performance of Sibyl:
   (a) How long to get the report from the project's issue repository server.
   (b) How long for Sibyl to make the recommendations.
   (c) How long to construct the augmented page.
5. When a questionnaire was given to the triager.
6. The results of the questionnaire.

Figure 4.5: Information logged by Sibyl.

Monitoring of Triage Activity

The subjects used an augmented version of Sibyl as part of their normal triage activities. Sibyl was augmented to log six pieces of information about the triager's activity (see Figure 4.5). As part of the user configuration of Sibyl, subjects consented to have this information logged and were assigned a unique study identification number so as to keep their identity and logged data anonymous. Subjects had full access to this logged information so as to be informed about what was being collected and were asked, not forced, to submit their activity logs weekly.

The logged information was recorded for a number of purposes. A time stamp of when a triager viewed a report was recorded for the purpose of determining how long it took the triager to make a particular change. A time stamp of when a change was made and the contents of the change request to the repository server were logged to provide the rest of the needed information to determine how long it took the triager to make a particular change, such as assigning the report to a developer. Also, from the change information, and recording what Sibyl had recommended for the report at the time the change was made, we determined the accuracy and recall for the recommenders. The performance of Sibyl was tracked so as to determine where, if any, were the bottlenecks in the service. Sibyl recorded when a questionnaire was given, and what the questionnaires responses were, to determine if the triagers were skipping the questionnaires.[14] Questionnaire responses were recorded for later analysis. To determine if appropriate numbers of recommendations were being presented to the triager, Sibyl recorded additional recommendations for each recommender. The additional recommendations were then used to perform experiments simulating Sibyl providing a larger number of recommendations. For R_A, Sibyl logged up to ten recommendations. For R_C and R_S, Sibyl recorded up to six recommendations, and

[14]We saw no evidence of triagers skipping questionnaires.

1. How long have you triaged for the Eclipse UI component?
2. What process do you follow when you triage? How do you approach the task?
3. What do you find is the easiest and hardest parts of triage?
   (a) What do you think would help make the hardest part less challenging?
4. Rank the four kinds of recommenders in order of most interested in using to least interested in using during your triage activities.
5. Are there other fields/information that you would like to see recommendations for?
6. What were your impressions of the Sibyl tool?

Figure 4.6: Questions for the post-Sibyl usage interviews.

for the R_I, Sibyl recorded up to fourteen recommendations. The triager was only shown either the top four (R_A), three (R_C and R_S), or seven (R_I) of these recommendations.

Subject Interviews

The four subjects were interviewed to gather more information about the project's triage process, more in-depth comments about using Sibyl and to gather more information about other ways that triagers could be assisted by recommendation in their work. Three of the triagers were interviewed after they had completed their triage turn and the fourth had just started using Sibyl. The time period between when the triager had used the tool and the interview ranged from a few months to currently using the tool.

Questions for the interviews were prepared in advance so as to focus the interviews. However, the specific questions asked to each subject depended on the flow of the interview and the subject's experience with Sibyl. For example, one subject was not asked about their impressions of using Sibyl for triage as they had not yet used the tool for a sufficient length of time. Figure 4.6 shows the prepared questions.

Methods of Quantitative Analysis

We used two methods for quantitative analysis. The first was used for determining if the recommenders work as well in practice as was indicated by the analytical evaluation (R4). We used the measures of recall and accuracy. The recall measure was the same as was used previously (see Section 4.1).

Accuracy measures if the recommender made the correct recommendation. For example, a R_A is accurate if it recommends the person that was assigned. We use accuracy instead of precision as it would be misleading to use the precision measure from Section 4.1;

Table 4.13: Accuracy of the recommenders from the field study.

	Measurement	Changes	In-Practice	Refined
Developer	Accuracy	270	75%	84%
Component	Recall	86	85%	94%
Interest	Recall	150	51%	72%

we did not ask the triager for an implementation expertise set for each reports.[15] It therefore could have occurred that the recommender suggested developers with the correct expertise, but the triager chose a different developer from the unknown implementation expertise set. This choice would cause the recommender to appear to have a lower precision than was true for that report.

To determine if the use of the recommenders reduced the time taken to triage reports (R2), we used an an independent t-test. As was stated before, Sibyl periodically designated report changes as control cases. As there was not enough data collected to do a by-subject analysis, we aggregated the data and performed an across-subject test.

Methods of Qualitative Analysis

For a qualitative analysis of the recommenders, we used the responses from the surveys and the interviews. We used this data to either confirm or refute the results found using the quantitative analysis and to broaden our understanding of how the recommenders affected the triage process.

4.2.3 Quantitative Results

Over the course of the study, we recorded that the triagers made 259 developer assignments, 86 component assignments, and 150 cc: list assignments using Sibyl over the four months of study.

Table 4.13 shows the results for the different types of recommenders from the field study. Presented are the accuracy of R_A and the recall for R_C and R_I. For each recommender we present two results: in-practice and refined. The in-practice result corresponds to how the recommender worked from the viewpoint of the triager, and the refined result refers to the result if cases where the recommender did not have correct information are removed from consideration.

[15]As one of our research questions was about the time taken to triage a report, asking for an implementation set for each report, an action a triager would not normally take, would have interfered with that portion of the field study.

Developer Recommendation Results

Over the course of the study there were 270 cases where an assignment was either explicitly or implicitly made.[16] To compute the R_A accuracy, we compared the four developer names recommended by Sibyl to the developers who actually resolved the report. We considered a correct recommendation had been made if one of the recommendations matched one of the individuals assigned to the report. The R_A analysis was done several months later to allow for assignment adjustments. A report may have its assignment adjusted in cases where the triager made an incorrect choice or a developer took over working on the report from another developer. Either of these cases would cause the report to be have been assigned to multiple developers. Taking the average over the 270 cases, the R_A achieved an in-practice accuracy of 75%. In other words, three quarters of the time the R_A made a correct recommendation. As we were using a component-based R_A, the recommender was selecting from nineteen different names.

There are two situations in which the R_A does not have the correct information to make appropriate recommendations. The first is if the report is filed against the wrong project (i.e., the report is not actually for the Eclipse product). The second is if the report is filed against the wrong project component. This second situation is a problem as the R_A used in the study made recommendations based on the value of the `component` field. If the `component` field was not correct, then the recommender will make recommendations from the wrong set of developers. We found that there were twenty-nine cases over the study period where either the report had been assigned to the wrong product or component or the triager had assigned the report to someone outside of developers known by the recommender. As all the triagers worked on the `UI` component of the `Platform` product, if either of these fields was changed at the same time as a developer assignment was made, we considered this to indicate one of the two situations. After removing these cases, the recommender's accuracy increased to 84%, the recommenders refined accuracy.

The version of Sibyl used by the triagers provided four developer recommendations to the triager. As we were uncertain if this value was sufficient, we logged ten recommendations in the background to determine if the recommender would have been more practical had more recommendations been provided (R4). We found that if the number of recommendations were raised from four to seven,[17] then the accuracy would have risen to 85% for the in-practice accuracy and 94% for the refined accuracy. It is not surprising that the accuracy would have improved had more recommendations been provided. That the average accuracy would have gone up by 10% had seven recommendations been made instead of four suggests there may be value in presenting more developer recommendations.[18]

[16]There were 259 cases where the triager assigned a report to a developer (i.e., explicit assignment) and 11 cases where the report was not assigned but was resolved (i.e., implicit assignment).

[17]The accuracy did not improve beyond seven recommendations.

[18]Making seven recommendations is less then showing the entire list of 19 developers for the

However, the increase in accuracy must be weighed against the additional effort required by the triager to search the presented names. Although an increase of 10% in accuracy is a significant improvement, it is unclear if this increase adequately compensates for the increased burden on the triager. Triager comments from the surveys confirmed that although four recommendations were reasonable, sometimes it was too few.

Component Recommender Results

During the field study, we recorded eighty-six component assignments. Given that the triagers were all from the same project component, this means that there were eighty-six reports that were incorrectly filed against the UI component and were reassigned to different components. Table 4.13 shows that for these reports, the R_C had an in-practice recall of 85%. In other words, the correct project component was one of the three recommendations 85% of the time.

There are two cases in which the R_C does not have the correct information to make an appropriate recommendation. The first is if the report is filed against the wrong product (i.e., the report is not for the `Platform` product). The second is if a new component is added to the project and the recommender does not yet have information about the component. We found that there were four cases of each of these situations over the duration of the field study. If these cases are removed from consideration, then the refined recall is 94%.

As with the R_A, we had made a guess that three recommendations was appropriate to present to the triager. However, we found that if the number of recommendations was raised to four, then the in-practice and refined recalls would have been 88% and 97% respectively. Again, it is not surprising that providing more recommendations results in a better recall, however that adding an additional recommendation results in such a small increase (3%) demonstrates that three recommendations is likely sufficient to provide a component recommender with high recall.

Interest Recommender Results

Over the course of the field study, we recorded 150 cases where the triager added names to the report's cc: list. Table 4.13 shows that the average in-practice recall for R_I was 73%. In other words, on average, R_I correctly recommended close to three-quarters of the people that appeared on the cc: list. For comparison, if we had instead recommended the seven most frequently occurring names on the cc: list from the data set, the in-practice recall would have been 48%.

However, we observed that triagers for the Eclipse project had a practice of adding themselves to the reports that they triaged. Taking this practice into account, there are forty-five cases where individuals other than the person making the change were added to

component.

the cc: list, and the refined recall is 60%. This result means that for these forty-five cases, R_I correctly recommended, on average, half of the individuals on the cc: list. Again, for comparison, if the top seven most frequently occurring names are used and the name of the person adding the names is removed from the cc: list, the recall is still 52%.

The analytical recall was found to be 49%, a much lower value than the in-practice recall and closer to the refined recall. This would seem to indicate that perhaps the triagers remove their names from the cc: list at some point after they have triaged the reports. If this is the case, then the practice of the triager adding their name to the report, which appears to be the reason for the high in-practice recall, would not have been evident in the data used for the analytical evaluation, and may explain why the refined recall is closer to the analytical recall.

We found that increasing the number of interest recommendations does not affect the in-practice recall or refined recall for R_I. This suggests that presenting seven interest recommendations is appropriate.

Triage Time Improvement

One of our research questions (R2) was whether or not the use of triage-assisting recommenders reduces the time taken to triage a report. To evaluate if R_{DO} recommenders do lower triage time, we collected information about how long it took a triager to make an assignment[19] when the four types of recommenders were available and compared this to the cases when no recommendations were made (i.e., control cases). The time to make an assignment was taken to be the time between when the triager first viewed the report and the time that they made an assignment (e.g., assigned to a developer or added a name to the cc: list). Recall that Sibyl designated every fifth set of recommendations as a control case and made no recommendations, providing a baseline.

For analysis, we did a paired t-test (i.e., repeated measures) of time taken for the triagers to make an assignment with and without a recommendation being provided. Table 4.14 shows the median values for each subject for the control cases and for the cases where the triager was provided recommendations.. We found that there was no statistical significance between the times taken to triage a report when recommendations were presented and when no recommendations were given to the triager at our chosen 95% confidence level (*t*=-1.098, *df*=3, *p*=3.53).[20] However, given that the statistical power of the test[21] is 12%, there is a 88% chance that if the recommenders affected the triage time, we would not have seen it. The lack of any significant effect may be caused by the group of control reports not being representative due to their relatively small number. For example,

[19]We use the term *assignment* to broadly refer to the result of a triager making a decision.

[20]A confidence level of 95% was chosen as this is the typical value used to determine statistical significance (See `www.socialresearchmethods.net/kb/power.php`).

[21]The statistical power measures the likelihood that if the recommenders had an effect on the triage time that we would have observed a change.

Table 4.14: Median values for control and treatment cases for recommendation.

	Control (minutes)	Recommendations (minutes)
Triager A	1.75	2.8
Triager B	1.01	4.34
Triager C	2.23	1.2
Triager D	1.58	2.18

one of the triagers had only one control case. Therefore, more study is needed to determine if the recommenders improve the time taken to triage a report, especially as we had such a small number of participants.

4.2.4 Qualitative Results

In this section we present the results from the decision and usage questionnaires, as well as information gathered by the triager interviews.

Survey Results

For R_A, the triagers found that the number of recommendations presented were either reasonable (5 responses) or too few (3 responses), confirming our observation from the quantitative analysis that four recommendations is a reasonable number to present, but that it may be helpful to let the triager decide how many recommendations are presented. The triagers indicated three times that multiple recommendations were applicable, but felt that choosing between them was very easy or easy.

For R_C, three responses indicated that the triagers felt that three component recommendations was too few, one response suggested that three recommendations was too many, and one response indicated that three component recommendations was reasonable. That the triagers tended to feel that not enough component recommendations were given is not consistent with the high top-three recall demonstrated by the logs. For the two times that a triager indicated that there were multiple applicable recommendations, the triagers again felt that it was easy to decide between them.

Only one triager submitted answers to the usage questionnaire. Recall that this questionnaire asked questions about the general usability of the recommenders. The triager commented that he[22] found the R_A to be very useful and the component recommender to not be useful.[23] In explaining why he felt the component recommender to not be useful he said:

> *Often my assignments in this space were between UI and IDE[24] and Sibyl didn't seem to be aware of IDE. Not unexpected, seeing as it's new.*

[22]We mix the gender so as not indicate which triager made these comments.

[23]The R_I was added later in the field study after this triager had used Sibyl.

[24]*IDE is another of the project's components.*

Table 4.15: Ranking of recommenders by triager interest.

Triager A	Triager B	Triager C	Triager D
Sub-component	Component	Sub-component	Sub-component
Component	Sub-component	Developer	Developer
Interest	Interest	Interest	Component
Developer	Developer	Component	Interest

The triager also felt that Sibyl made him consider other alternatives that he would not have otherwise considered and, with the exception of the component recommender, there were no differences between the recommendations and his expectations. The triager also felt that Sibyl made it faster to do assignments.

Interviews Responses

Table 4.15 shows the rankings of the four kinds of recommenders by the four triagers. Note that the R_S is consistently ranked near the top across the four triagers. When asked to elaborate on their ranking, the triagers all stated that if the R_S is correctly determined, that this dictated the developer assignment (hence the low ranking of the R_A by the first two triagers as they felt the recommender was redundant). A similar reason was given for the high ranking of R_C. The triagers also consistently felt that R_I was of lesser value than the other recommenders, when taking into account that the triagers viewed the R_S, R_C, and R_A as providing equivalent functionality.

The information provided about the triagers experience was already given in Section 4.2.1 and the triager's responses about their triage process were given in Section 2.1.1. We defer the discussion of what other fields and information the triagers would like to have had recommended to Section 8.1 where we discuss future work.

Comments about Sibyl

One of the triagers commented that he felt that the R_A had a high accuracy. Another triager commented that she felt that Sibyl gave good recommendations, but would sometimes get confused. For example, if the report contained a stack trace, Sibyl would recommend the SWT[25] component due to a SWT thread exception occurring in the stack trace. In other words, Sibyl would make a recommendation based on the effect of a fault, not the cause of the fault.

One of the problems with Sibyl that was pointed out by the triagers was that it made Bugzilla feel less responsive. One of the triagers plainly stated that using Sibyl was slower than using Bugzilla directly. Given the architecture of Sibyl, this comment is not surprising; Sibyl redirects Bugzilla requests and would magnify any responsiveness problems

[25]SWT is an open-source widget toolkit for Java that is used for the Eclipse user interface and is one of the project components. See `www.eclipse.org/swt` for more details (verified 26/06/07).

Table 4.16: Performance of the Sibyl service (in seconds).

	Min	Average	Max
Get Report	0	3.25	47
Make Recommendations	0	0.7	43
Augment Report Page	0.8	2	11

of the Bugzilla server.

4.2.5 Sibyl Performance Results

As mentioned in Section 4.2.2, we recorded some performance measures about how Sibyl worked. Specifically we recorded how long it took to get a report from the server, how long Sibyl took to make a recommendation for the report, and how long it took to construct the augmented report web page. Table 4.16 shows the minimum, average, and maximum time in seconds taken by Sibyl to perform these three actions. On average, Sibyl takes very little time to make a recommendation. The exception is when the server has been rebooted or the recommenders have been updated and the recommenders need to be loaded into memory. As the recommenders can range in size up to a hundred megabytes, this loading can be time consuming. As the recommenders are updated daily, we eliminated this delay by having the server load the recommenders into memory as part of the update process. Once this bottleneck was removed, we found that the most time consuming part in using Sibyl was the retrieval of the report from the server. Since the bottleneck is the report retrieval, if Sibyl were installed on the same server as the bug repository, it would have a minimal performance impact on the server during its use as it could directly query the issue repository database for the relevant information.

4.2.6 Threats to Validity

The first threat to the validity of the results of the field study is that it was only conducted on a single project. Similarly, all of the participants worked on the same component of the project. Therefore, the results that the recommenders have a high in-practice accuracy or recall may not extend to other projects or components of the same project. This lack of generality may certainly be true for R_A as we used a component-based R_A for the field study and an analytical evaluation of this kind of recommender showed it to work best for the Eclipse project and not as well for the other projects (see Section 6.2.1).

A second threat to the field study results is the small number of participants. These triagers may not be representative of an average triager. All the participants were also developers for the project and therefore had access to project knowledge that volunteer triagers may not have. Having a larger number of participants across multiple projects would have reduced this threat.

The timing of the field study may have affected the results. The study was conducted in the middle of a development cycle, yet analysis of the Eclipse project has indicated that the average number of reports the project receives rises in the three months prior and post to the release of the product [3]. This timing may account for the lack of sufficient data to test our hypothesis that the recommenders improve triage time. However, it is also less likely that the triagers would have been willing to try the tool during the pre-release crunch.

The effect that the Sibyl tool had on the triager's workflow is unclear. Although attempts were made to provide recommendations in such a manner as to not adversely affect the triager's normal workflow, such as inserting the recommendations into the report web page as opposed to providing an external tool, there were aspects of data collection process that may have interfered. The two items that may have interfered the most are that a control case occurred every fifth change and that a survey was given after every tenth change. Despite having been forewarned that Sibyl would periodically not make recommendations (i.e., insert a control case), it is possible that triagers would still be caught off guard when it occurred, and may have led to a level of distrust with recommenders as the triager would expect a recommendation and not receive one. We did not see any evidence that this distrust occurred. The surveys did seem to interfere with triager workflow as one of the triagers stated this during an interview.

Another possible threat to the validity of the field study is that the triagers automatically accepted the top recommendation without considering if the recommendation was appropriate. This blind acceptance of recommendations could have led to the high R_A accuracy results. However, we doubt that this was the case because all of the triagers had over four months of triage experience for the project as well as being developers for the project and would be more likely to rely on their own experience than on the tool. Also, the analysis of the triager assignments was conducted several months afterwards to account for assignment mistakes (i.e., if the triager made an incorrect assignment it was likely corrected by the time the analysis was done).

Due to the manner in which the data was collected, it is unclear how much the triagers actually used the recommenders. As we collected only the results of a triager's action, such as a change to a report, and not the triager's interaction with the tool, we only know what changes were made, but not how they were made. It may be that the recommenders were good, but that the triagers never used them beyond a cursory trial. Another consequence of how the change information was collected is that we inferred what type of change was made post-trial using the report's history, which could lead to inaccuracy in the result with respect to time taken to triage a report.

There were inconsistencies in how long the triagers used the tool. In all cases the triagers had to be contacted individually to use the tool after it became their turn to triage for the project. There was therefore no immediate continuity between triagers. This lack of continuity resulted in some triagers using the tool for only a week of their turn and others using it for most of their turn and may have led to an inconsistent view of the triagers activity.

As with all field studies there is the threat of experimenter bias. However, as we based the in-practice and refined results on the logged data and the report history, we feel that there is little bias in these results and in how long triagers took to triage a report. It is however possible that experimenter bias affected the wording of the surveys and led to the more favourable survey responses than are actual true. However, as the follow-up interviews did not reveal this type of discrepancy we do not think this is likely.

4.2.7 Summary of Field Study Results

We conducted the field study to answer four questions.

R2: **Do recommenders created using ML$_{Triage}$ reduce the time it takes to triage a report?** We found that there was no statistical difference between the cases when triagers were given recommendations and when they were not. However there was insufficient statistical power to make a conclusion and more study is needed to answer this question.

R3: **Do the recommenders created using ML$_{Triage}$ help triagers to make better decisions?** This question largely pertains to the use of the recommenders by novice triagers. As the four study subjects were all expert triagers for the project, we were unable to collect the necessary data to answer this question. As mentioned in Section 4.2.1, we tried to recruit triagers from projects known to have novice triagers to answer this question, however we were unsuccessful.

R4: **What is the empirical performance of recommenders created using ML$_{Triage}$?** We found that the R_A had an in-practice to refined accuracy range of 75% to 84%. Although these results are not directly comparable to the precision values from the analytic evaluation, the results from both evaluations seem to indicate that R_A would assist triagers, especially those that do not have a deep project knowledge.

We found that the R_C had an in-practice to refined recall of 85% to 94%. Compared to the 92% recall found in the analytic study, the field study confirms that the component recommender can assist triagers in assigning reports to the correct project component and may be useful as part of the report submission process.

Finally, we found that the in-practice recall of R_I was 73%. This result is better than the result from our analytical result of 49%, and may be due to the relatively smaller number of field study test cases (150 vs. 860 in the analytical evaluation).

Since the refined recall for R_I (60%) was closer to the recall found in the analytical evaluation (49%), the triagers may remove themselves from the cc: list of the reports they triage at some point. As we do not know who triaged which reports in the data set used for the analytical evaluation, we cannot confirm this.

R5: **What is an appropriate number of recommendations to present to**

the triager?

We found that making four recommendations for R_A was generally sufficient, but that sometimes the triagers wanted to have more recommendations. For R_C, we found that presenting three recommendations was an appropriate number. Although more recommendations for R_I may have been more useful, we feel that the additional burden of increasing the number of recommendations from seven to fourteen would not be compensated for by the extra triager effort that would be required.

Chapter 5

Assisted Configuration of Recommenders

A triager who wishes to benefit from recommenders created using the machine learning-based triage-assisting recommender creation process (ML$_{Triage}$) faces at least two challenges in applying ML$_{Triage}$ to the data from his or her project. First, the data extracted from the bug repository is often noisy and must be cleaned for the technique to work effectively. This cleaning must typically be performed by a human expert from the development project. For example, for a developer assignment recommender (R_A), the list of developers who appear in the source repository must be cleaned to remove the names of developers who are no longer working on the project (see Section 3.2.5). Second, using ML$_{Triage}$ requires a substantial amount of up-to-date information from the repositories (e.g., eight months worth of bug reports), necessitating that the technique run either as part of the repository or on a mostly complete duplicate of the repository. Until the first challenge is solved, it is not possible to easily insert the technique into the repository infrastructure. Creating a parallel repository with cleaned data is possible but problematic because of the increased administrative load and infrastructure needed to keep the repositories synchronized.

In this chapter, we present an approach to address both of these challenges. We introduce a semi-automated configuration process that enables a human expert for a project to easily configure a project-specific development-oriented decision recommender (R_{DO}), such as a developer assignment recommender (R_A), a product component recommender (R_C) or an interest recommender (R_I). We also show that it is feasible to create a recommender using a subset of the data in a repository. By running over a subset of the data, it becomes possible to run the approach client-side, instead of server-side, easing the introduction of R_{DO} recommenders into a particular project's triage process. We demonstrate the configuration process and the effectiveness of the data subset approach in the context of a R_A. Specifically, we show it is possible to configure R_A that achieves within 10% of the precision rates of the R_A presented in Chapter 4 using only 27% to 44% of the reports for the five projects. We conclude this chapter by providing a description of how the assisted configuration approach can be used to create both R_C and R_I.

5.1 The Assisted Configuration Approach

Consider that you are a triager who wishes to create a R_A for your project. From prior work creating R_A using ML_{Triage} (i.e., Chapters 3 and 4) you know that R_A with high precision have been created using the Support Vector Machines algorithm, assigned and resolved bug reports and the text from the summary and description of the reports. However, to create the project-specific R_A you still need to answer the questions of how to label the reports (i.e., what are the labeling heuristics for the project?) and which of the developers are appropriate for recommendation. You wonder if it would be possible to use fewer reports than the eight-months worth of reports used in the previous work to create the R_A.

To aid the triager in the creation of the project-specific R_A, we describe an approach in which data from the report repository is sampled, summarized and presented to the user to help make the project-specific decisions quickly. The approach assists the triager by first presenting a subset of the repository data to help specify the project heuristics. Figure 5.1 shows a mock-up of an application that assists the triager in specifying heuristics by representing groups of reports by regular expressions and allowing the triager to state what report information to use for report labeling. The potential user interface is described in more detail in Section 5.1.1.

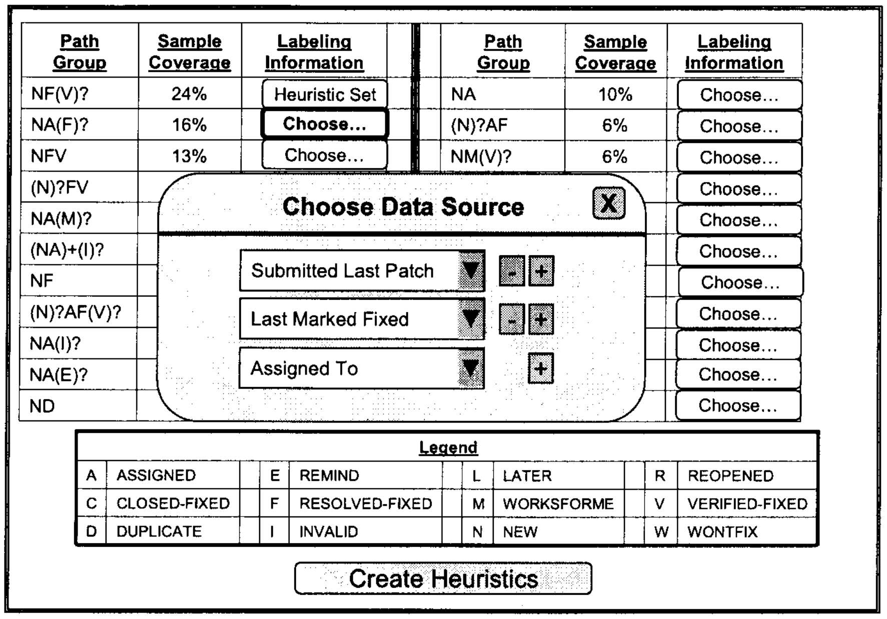
Once the triager has specified the heuristics, the approach applies the heuristics to a subset of the repository data to present the triager with a graph showing the report resolution activity for the developers and suggesting an activity threshold.

The final step of the approach assists the triager in determining an appropriate sized subset of reports to use for creating the R_A. Through iteratively creating R_A with increasing amounts of data until the precision plateaus, the approach produces a suggestion of the amount of data to use for creating a R_A for the project.

5.1.1 Selecting and Labeling the Reports

To assist the user (i.e., the triager) in creating the project-specific heuristics, we first extract a sample of reports from the repository. We then present the user with summary information about the reports in the sample. The summary information describes different groups of reports and we refer to these groups as *path groups*. For each of these path groups, we also present the user with a summary of information from the reports in the group that could be used to label these reports. The user then selects the report groups that he or she wishes to use for training the recommender by indicating the report information to use in labeling the reports for the group.

Recall from Section 3.2.4 that we chose to establish the labeling heuristics for a R_A by examining the logs of a random sample of reports from the project to create the heuristics,



Path Group	Sample Coverage	Labeling Information	Path Group	Sample Coverage	Labeling Information
NF(V)?	24%	Heuristic Set	NA	10%	Choose...
NA(F)?	16%	Choose...	(N)?AF	6%	Choose...
NFV	13%	Choose...	NM(V)?	6%	Choose...
(N)?FV					Choose...
NA(M)?					Choose...
(NA)+(I)?					Choose...
NF					Choose...
(N)?AF(V)?					Choose...
NA(I)?					Choose...
NA(E)?					Choose...
ND					Choose...

Choose Data Source

Submitted Last Patch

Last Marked Fixed

Assigned To

Legend

A	ASSIGNED	E	REMIND	L	LATER	R	REOPENED
C	CLOSED-FIXED	F	RESOLVED-FIXED	M	WORKSFORME	V	VERIFIED-FIXED
D	DUPLICATE	I	INVALID	N	NEW	W	WONTFIX

Figure 5.1: Mock-up for heuristic configuration.

Table 5.1: Legend for bug report life-cycle states.

U	UNCONFIRMED
N	NEW
A	ASSIGNED
F	RESOLVED-FIXED
V	VERIFIED-FIXED

but that an alternative is to derive the heuristics from direct knowledge of the project's process. By presenting a user with information about the project, we are employing this alternate technique.

Grouping Reports by Life-cycle

To characterize reports in the repository, we randomly sample reports at intervals of one, two, four, and eight weeks in the recent past. We take this approach to ensure we have a sufficient representation of the current report activity. A sample for an interval consists of twenty-five reports, resulting in a maximum total sample of 100 reports.[1] The report sample is taken from the set of reports that are marked as being in certain states as indicated by the user, such as reports marked as ASSIGNED or RESOLVED.

The next step in the characterizing the reports is to group the sampled reports by similarity in their life-cycles.[2] We chose this approach because label assignment depends upon the actions performed to the bugs, such as whether or not a patch has been submitted

[1]The size of 100 reports is approximately 2 MB for the projects we have analyzed.

[2]See Section 2.2.2 for more information about bug report life-cycles.

for the bug. We determine the life-cycle path of each report from its history log. We represent each life-cycle path as a string with the characters in the string representing each state a report has passed through (see Table 5.1 for a mapping of characters to states that we use). We now want to group the reports with similar life-cycle path strings. We chose to form these groupings by deriving regular expressions to represent similar life-cycle paths. Determining a regular expression from a set of examples, as we need to do to form path groups, is known to be a hard problem. The survey paper by Sakakibara on grammatical inference [56] lists several bodies of work that provide computationally hardness results for this process. Consequently, we used a heuristic approach derived from observations of common path patterns found in a variety of projects and that involve simple transformations of the path expressions.

After creating the regular expressions from the life-cycles of the sampled reports, we collect the regular expressions into a set. The regular expressions are then applied to the paths from the sampled reports.

Our approach to determining and ranking path groups is similar in intent to those used in sequential pattern mining [1]. Sequential pattern mining is the process of determining commonly occurring series of transactions in a database, such as people who rent "Star Wars" then rent "The Empire Strikes Back". Sequential pattern mining is similar to our approach in that we too are mining for commonly occurring state patterns in the life-cycles of bug reports and ranking them. However, sequential pattern mining ignores transactions that are in between other transactions, such as renting "Jaws" before renting "The Empire Strikes Back". As a result, sequential pattern mining is inappropriate for our use as we cannot ignore intermediary states in the path. Also, sequential pattern mining does not capture cycles such as a report moving back and forth between two states.

Forms of Path Regular Expressions As previously mentioned, we create four forms of regular expressions from the life-cycle paths of the reports in sample. The four forms are:

1. the path expression itself,
2. path expressions containing a cycle,
3. path expressions that exclude a life-cycle state at either the beginning or end of the path, and
4. a combination of the second and third form.

These regular expression forms increase in their level of generality. For example, a path expression with a cycle is a more general form than the path expression itself. The use of more general path expressions allows the user to specify heuristics at the level of generality he or she wishes to use. For example, the user may want to specify a heuristic for reports that follow the path NAF and a different heuristic for reports that follow the path NAFV, or, he or she may want to just specify a heuristic for both of these groups using the more general expression NAF(V)?.

The first regular expression form is the trivial expression of the path itself. This expression represents the most specific path grouping. The next form captures cycles in the path. For example, the regular expression (NA)+F matches paths that alternate between the NEW state and the ASSIGNED state before ending in the RESOLVED-FIXED state. Paths are examined for cycles of prime sizes ranging from two to half the length of the path.

We observed a common occurrence with projects that have two states for representing a new report (e.g., UNCONFIRMED and NEW). Some reports start in the first new state and move to the second state, and some reports just start in the second new state. For example, the Firefox[3] project has one state to identify reports that have been recently submitted but for which the problem has not been verified (UNCONFIRMED) and another state to indicate that the problem has been verified but not yet been fixed or assigned (NEW). If a Firefox developer submits a report, she will mark the report directly as NEW because she will have verified that the problem exists. A similar case occurs where reports that are resolved as being FIXED do not always get marked as VERIFIED. As we view these kinds of reports as following similar paths, we use a third regular expression form to capture the exclusion of one state at either the start or end of the path. For example, the expression (U)?NF represents reports that follow the path NF with or without starting in the UNCONFIRMED state. Similarly, the expression NF(V)? represents reports that follow the NF path with or without ending in the VERIFIED-FIXED state. We also examined the use of regular expressions that capture the exclusion of two states at the start or end of the path (e.g., (U)?(N)?F), but we found that this produced many regular expressions that were not meaningful and thus we do not use it.

The final regular expression form consists of various combinations of the second and third forms, such as (U)?(NA)+ and (U)?(NA)+F(V)?. The first expression is for the path group representing reports that have a NA cycle and may or may not have started in the UNCONFIRMED state. The second example represents the path group for reports that may or may not start in the UNCONFIRMED state, alternate between the NEW and ASSIGNED states before moving to the RESOLVED-FIXED state, and possibly the VERIFIED-FIXED state.

Path expressions that match only one or two paths are deemed to be too specific and are removed from the set of expressions. For the five systems we have tested (see Section 5.2), this process resulted in a range of 13 to 33 expressions. This list size is tractable for a user to scan to identify path groups of interest.

[3]Firefox is a web browser and can be found at `www.mozilla.org/products/firefox` (verified 06/06/07).

Determining Data Sources

Given the groupings of reports, the user now needs to describe how to label each group of reports. Before a labeling heuristic can be specified by the user, we need to determine what information from a report can be used to label it. We have found that it is not possible for one rule to fit all reports. For example, one of the heuristics for the Firefox project uses different report information depending on if there was an approved patch and the number of individuals that had submitted approved patches (see Appendix A). Thus, we use the path groups representing different kinds of reports as the basis for this specification. We refer to the pieces of data that can be used to help label a path group as *data sources.* Examples of data sources include the value of the `assigned-to` field or the person who attached a patch to the report.

The data sources that can be associated with a particular path group are determined from reports that belong to the path group. The fields and history log for these reports are mined for occurrences of user names.[4] For example, Figure 5.1 shows that the analysis of the reports in the NA(F)? path group has determined at least three data sources: an attachment-added event in the log ("Submitted Last Patch"), a resolved-fixed event in the log ("Last Marked Fixed"), and the value of the `assigned-to` field ("Assigned To"). Not all data sources may be available for each member of a path group; as a result, a user may select multiple data sources for a single path group. For example, not all reports that are in the path group represented by NA(F)? will have a data source stating who resolved the report as fixed; a user might thus specify both the resolved-fixed event and the `assigned-to` field, as shown in Figure 5.1. The particular data source to be used for a particular report depends on the order in which the heuristics are applied during labeling. For the heuristic being specified in Figure 5.1, the data sources for labeling a report will be used in the order of attachment-added event, resolved-fixed event, and `assigned-to` field.

Example: Specifying Labeling Heuristics

Given the path groups and the data sources, it is possible to specify the labeling heuristics. Figure 5.1 shows a mock-up of an application for assisting the user in specifying these heuristics. The application presents three pieces of information to the user. The first piece of information is the regular expressions describing the path groups (Path Group column).

The second piece of information provided is a coverage metric for each regular expression (Coverage column). The coverage metric indicates how many of the reports, as a percentage, in the sample are members of the path group. For example, in Figure 5.1 24% of the reports in the sample are members of the NF(V)? path group. The regular expressions are presented in decreasing order of coverage. This ordering directs the user

[4]The mining of user names is specific to the creation of a R_A. Other recommenders mine different data (see Section 5.3).

toward creating labeling heuristics for commonly occurring reports.

The final piece of information presented to the user is the sources of data that can be used for labeling the reports of the path groups. As shown in the figure, each regular expression is associated with a button. The button opens a wizard that is used to specify the data sources to be used for labeling reports in the path group. The wizard initially contains a single drop-down box listing the data sources that can be used for labeling the reports in the path group. The user can also add additional data sources by clicking on the '+' button to the right of the drop down box (or use the '-' to remove a data source). The figure shows that the user has already specified data source(s) for the NF(V)? path group and is in the process of selecting the data sources for the NA(F)? path group. The user has selected to use two pieces of information from the report's history log: who last attached a patch to the report and who last marked the report fixed. The user has also selected to use the value of the `assigned-to` field if neither of these events are present in the report's history.

5.1.2 Selecting the Labels

As described in Section 3.1, not all information in the repository is useful. For instance, for a R_A, not all developers listed in the repository are currently active on the project. We thus need to remove these developers from consideration, which corresponds to selecting the labels to use in the recommender.

To determine the labels for a R_A, namely the set of active developers, we present the user with a graph showing a distribution of report resolution by developer. This graph is produced by labeling all resolved reports for the most recent three months with the user-specified heuristics and ordering the developers from most resolved to least resolved. We use the median of the probability distribution that typically best fits a report resolution curve, a Pareto distribution, to suggest a threshold to the user for a cut-off activity level.

Our decision to use a Pareto distribution to model report resolution activity is based on producing the distributions for the five projects we examined.[5] We found that the Pareto distribution was consistently ranked near the top by the Anderson Darling test.[6] Intuitively, we feel that this choice makes sense as the Pareto distribution is the basis for the '80-20 rule' that states that 80% of the work is done by 20% of the individuals, and this trend has been observed with other software artifacts [14, 40, 46, 54].

Since not all project's report resolution activity will fit a Pareto distribution and because a user may have better knowledge about where to set a cut-off, we allow the user to change the threshold after viewing the distribution. Figure 5.2 shows an example of a user setting the threshold for project differently than the median. In the figure, a threshold

[5]See Section 5.2 for a listing of these projects.

[6]The Anderson Darling test is a common test for assessing if a data set comes from a particular probability distribution.

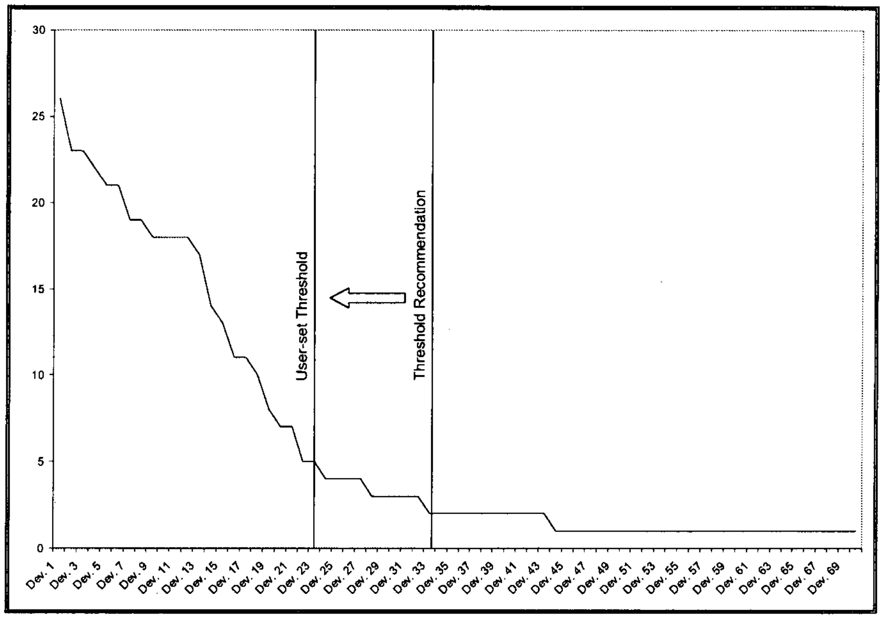


Figure 5.2: Example of setting the reports-resolved threshold for selecting the active developer set.

of two reports resolved has been suggested, but the user has chosen to move the threshold up to five reports.

We use all of the labeled data for three months to get an accurate portrayal of the developers' activity. Although taking a random sample over this time period would use less data, we feel the risk of misrepresenting developers is too great, especially as this profile is used in selecting the training data. For the five projects on which we have applied this approach (Section 4.1), this data amounts to approximately 2 to 3 MB.

5.1.3 Selecting the Reports

A machine learning algorithm must be trained with sample data. In the specification of the heuristics, the user begins the process of selecting which reports to use to train the R_A. For example, the user may choose to provide labeling heuristics only for reports that end in the RESOLVED-FIXED state. The user may also choose to include those reports that have been marked as DUPLICATE. Or the user may choose reports that have been marked as WONTFIX as the user knows that for the specific project only developers with particular expertise relative to the area of the report are involved in this decision.

To complete the selection of training data, it also must be determined how many of the reports to use. Most recommender techniques follow the approach we have used previously; the more training data the better. Although this approach is reasonable in a server environment, this approach is not typically practical for a client-side setting. Instead, we want to be able to determine the right amount of data to use for a particular project automatically.

Work by Forman and Cohen supports our hypothesis of being able to use less data. Also, Dietterich noted that "as more data is available, the [classifier] accuracy reaches a higher level before eventually dropping." [24, page 5]. Forman and Cohen investigated the performance of various machine learning algorithms using small training sets for binary text classification tasks [29]. They found that both Support Vector Machines and Naïve Bayes worked reasonably well with partial data sets.

To discover the right amount of data for a particular project we use a series of automated experiments whereby an amount of randomly selected data is used to create a recommender and the amount is gradually increased until the accuracy of the produced recommender plateaus.

Once the trial data is collected, we use stratified five-fold cross validation [68] to determine the quality of the recommender using this amount of data. Stratified five-fold cross validation is the process whereby the data is randomly divided into five groups with equal representation of each developer in the groups. A recommender is then created with four-fifths of the groups and the last group is used for testing the recommender. This procedure is repeated five times with each group serving as the test set. The accuracy of the recommender is the percentage of how many test reports for which the recommender correctly predicted the label.[7] Recall that the reports are labeled using the user-defined heuristics. The algorithm used to create the recommenders is the Support Vector Machines algorithm based on the results from Chapter 3.

The accuracy of a particular data level is then compared to the accuracy of the previous two data levels. If the average difference between these values is less than 1%, we conclude that the accuracy curve has plateaued and we take the minimum of the three data levels as a potential project data level. As this process involves randomness in the partitioning of the data for cross validation and randomness in selecting the reports that are in the developer report samples, we repeat this process five times and take the median of these trials as the number of reports to sample for the project to train the recommender.

As we are wanting to determine the amount of data we need for recommender creation, in each iteration we select more data than the amount that we are evaluating. For example, if we are testing to see if 100 reports is sufficient, a total of 120 reports are selected so that in each fold of the cross validation, 100 reports are used for recommender creation. As the data is selected per developer, the number of reports actually selected for the developer is increased by the appropriate amount.

[7]We use accuracy and cross-validation in this situation because we are focusing on the selection of the appropriate amount number of reports. We chose not to use stratified five-fold cross-validation for evaluating and comparing the performance of the recommenders in Chapter 4 because the precision and recall measures require information about all the possible labels (i.e., the set of developers with the appropriate experience) and cross validation uses only one correct label value.

Table 5.2: Date ranges for training data and testing set sizes used in evaluation.

	Start Date	End Date	Testing Reports
Eclipse	Oct 1, 2005	May 31, 2006	152
Firefox	Feb 1, 2006	Sept 30, 2006	64
gcc	Apr 1, 2006	Nov 30, 2006	73
Mylyn	Feb 1, 2006	Sept 30, 2006	50
Bugzilla	Feb 1, 2006	Sept 30, 2006	52

5.2 Evaluation of the Approach

We evaluate our assisted configuration process along three dimensions. These dimensions correspond to the three configuration stages presented in Section 5.1: the heuristics used, the threshold for selecting active developers, and the data selection strategy used for tuning and training the recommender. We perform the evaluation with five projects to gauge the generality of our configuration approach. We use the same five projects as in Chapter 4: Eclipse Platform, Firefox, gcc, Mylyn, and Bugzilla. We compare the results of the R_A recommenders produced for each project to the R_A recommenders created using ML_{Triage} from Chapter 4.

As in Chapter 4, we used assigned and resolved reports from an eight-month period as the data set from which reports are sampled and drawn for creating a R_A using the assisted configuration approach. The reports used for testing the created recommenders were taken from the following month for which we could find information in the source repository. Table 5.2 shows the date ranges and the size of the testing sets for the five projects. Further details of the recommender configurations are given in Appendix B.

5.2.1 Heuristics Used

The first dimension evaluated is the heuristics used. We experimented with this dimension to understand how many heuristics a user would need to specify to get a good recommender. We investigated this by evaluating recommenders created when specifying heuristics for the top five and top ten presented path groups. In setting the data sources for the heuristics, we used domain knowledge acquired from constructing the heuristics in Chapter 4. In one case, this resulted in our not setting a data source for some of the path groups of the Firefox project. Specifically, for this project some of the top ten path groups were for reports that moved from UNCONFIRMED to INVALID. As we had previously observed that these reports are intercepted by triagers, they do not contain information about which developer would have fixed the problem and cannot be assigned a label. This knowledge would be obvious to a Firefox project member.

Table 5.3: Active developer thresholds used for evaluation.

	Suggested		User-defined	
	Threshold	Devs.	Threshold	Devs.
Eclipse	2	48	5	41
Firefox	2	69	5	36
gcc	2	35	5	18
Mylyn	3	7	10	6
Bugzilla	2	13	5	7

5.2.2 Threshold Setting

The second dimension evaluated is the active developer threshold. We experimented with this dimension to gauge how sensitive the recommender creation process is to changes in the threshold. We investigated this dimension by considering the results when the suggested threshold is accepted and when the user specifies a threshold. Table 5.3 shows the values that we used and the number of developers in the recommendation set that result for each of the five projects. The user-specified values were determined by examining the report resolution distribution and judging the point at which the curve started to flatten.

5.2.3 Data Selection Strategy

The final dimension we investigated was the effect of using three different data selection strategies for tuning and training the recommender. We experimented with different strategies in order to find one that selects just enough data to make a good recommender. The first data selection strategy randomly selects a fixed number of reports. In each subsequent tuning iteration this number of reports is increased until the tuning process halts. This value is then used to train the recommender. In our experiments with this data selection strategy we began the tuning with 100 reports and the size was increased by 100 in each iteration. We refer to this data selection strategy as "random".

One possible problem with the random strategy is that it may not reflect the relative contributions of the developers. In other words, it is possible that few reports are chosen for a prolific developer and vice versa, causing a recommender to make more incorrect recommendations. Therefore we experimented with a data selection strategy where the data is selected so that each developer is represented proportionally according to their relative contribution to the project. The distribution of developer contribution was normalized so that the developer contributing the least had a value of one. The tuning process proceeds by iteratively incrementing the factor by which the developer contribution distribution is multiplied. In other words, in the first iteration, the lowest contributing developers has one report in the data set, has two reports in the next iteration, and so forth until the total number of reports for each developer is reached. Whereas the contribution distribution provides the number of reports to use for each developer in the dataset, the reports

Table 5.4: Overview of recommenders created using ML_{Triage}.

	Threshold	Data Amount	Developers	Precision	Recall	F-Measure
Eclipse	9	6569	43	76%	13%	22%
Firefox	9	2621	39	73%	1%	2%
gcc	9	1791	28	82%	3%	6%
Mylyn	9	683	6	98%	30%	46%
Bugzilla	9	745	5	96%	4%	8%

themselves are selected randomly. We refer to this data selection strategy as "strict".

The last data selection strategy is a variation of the strict proportional strategy. Instead of adhering strictly to the developer's relative contribution, the reports are selected randomly until each developer has their proportional number of reports ±25%. This strategy is referred to as "tolerant" and is intended to provide a comprise between the random and strictly proportional strategies.

5.2.4 Results

We use two metrics to evaluate the effectiveness of the created recommenders: precision and recall (Equations 4.1 and 4.2 respectively). The details of how we compute these metrics using information extracted from the source repository can be found in Chapter 4. Although we present both values for completeness, we focus on precision because we believe it is more important to recommend a small set of right developers than a large set with many incorrect recommendations. Our evaluation considers a recommendation set of size one. Table 5.4 reports the values of these metrics for the five projects from Chapter 4.

As there is a level of randomness to the selection of the reports that are used to train the recommender, it is rare that the training process will produce the exact same recommender across different applications of the process to the same data. Consequently, the results in Tables 5.5 through 5.7 are the average of ten trials.

How Many Heuristics Do We Need?

The data in Tables 5.6 and 5.7 demonstrate that the use of the top five heuristics is sufficient to produce a recommender with good precision. As would be expected, the use of even more heuristics generally produces a recommender with even better precision as more heuristics provides for a larger set of reports for sampling and training.

A general guideline to projects is to use ten heuristics. If the top several heuristics provide high coverage of the reports, as is the case for the Mylyn project, it may be sufficient to specify fewer than ten heuristics. Either way, it is tractable and practical for a project member to specify up to ten heuristics.

How Sensitive To Threshold Changes?

Table 5.6 shows that the selection of the threshold to use for determining an active developer does not typically have a large effect on the created recommender.

The setting of the threshold value can sometimes cause a significant difference in the amount of data needed for the proportional data selection strategy. This effect is best seen with the Mylyn project in Table 5.7. If the user-selected threshold of ten is used, 199 reports are selected for training. If the suggested threshold of three is used, 508 reports are selected. Although the difference in the number of developers considered between these two threshold settings is one, the added developer has such a low contribution level that the normalization process causes many more reports to be selected. Feedback should be provided in a user-interface supporting assisted configuration for a recommender to allow a user to make the appropriate trade-off.

How Much Data is Needed?

The results in Tables 5.5 to 5.7 demonstrate the tolerance and stability of recommenders produced using the assisted configuration process. In most cases, the produced recommenders are within a few percentage points of the recommenders created with ML_{Triage} (Table 5.4). This data also shows that good recommenders can be produced using less data. For example, configuration based on the top ten heuristics, a user-selected threshold, and the proportional data selection strategy creates recommenders within 10% precision of those created with ML_{Triage} based on only 27% to 44% as many reports.

The plot in Figure 5.3 provides a different view of the data from the tables, showing how much data was used for the different data selection strategies and the precision that resulted. This plot shows that the random strategy ended up using all of the data for four of the five projects (far right of the plot). This plot also shows that the strict strategy, which selects reports strictly according to each developer's relative contribution, is better at producing good recommenders with less data. This result is evidenced by more of the strict data points being to the left in the plot than those of the other two strategies.

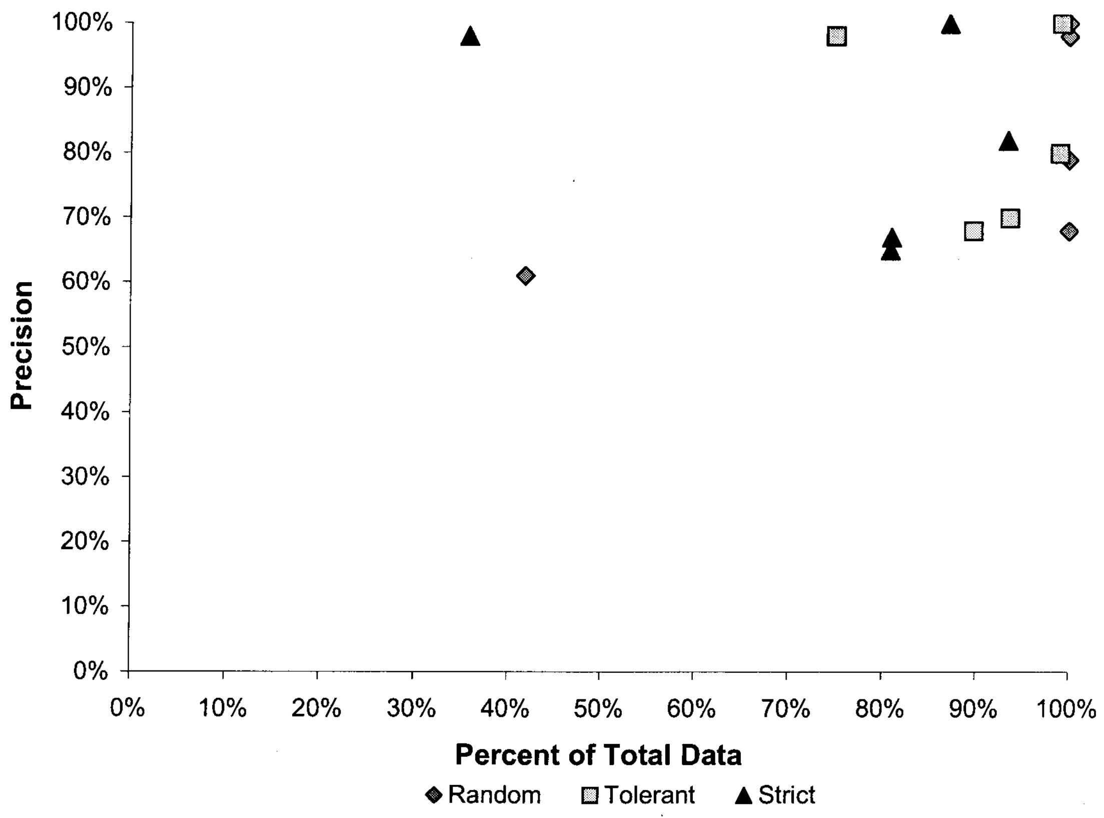


Figure 5.3: A scatter plot of data size vs. recommender precision.

Table 5.5: Precision and recall for data selection strategies for tuning and training a developer recommender.

Strategy		Random		Proportional-Tolerant			Proportional-Strict		
	Total Reports	Data Size	Result (P/R/F)	Data Size	Iter.	Result (P/R/F)	Data Size	Iter.	Result (P/R/F)
Eclipse	2861	1200 (42%)	61/10/17	2568 (90%)	7	68/12/20	2318 (81%)	7	67/12/20
Firefox	1025	1025 (100%)	68/1/2	960 (94%)	9	70/1/1	829 (81%)	7	65/1/2
gcc	518	518 (100%)	79/3/6	513 (99%)	15	80/3/6	484 (93%)	11	82/3/6
Mylyn	556	556 (100%)	98/30/46	417 (75%)	19	98/31/46	199 (36%)	9	98/30/46
Bugzilla	374	374(100%)	100/6/11	371 (99%)	10	100/6/11	326 (87%)	7	100/6/11

Table 5.6: Configuring a developer recommender with different heuristics and thresholds using all the possible data.

Heuristics	Top 5				Top 10			
	Data Size	Sug. (P/R/F)	Data Size	User (P/R/F)	Data Size	Sug. (P/R/F)	Data Size	User (P/R/F)
Eclipse	1722	62/10/17	1650	63/10/17	2907	72/13/22	2861	72/13/22
Firefox	1270	64/1/2	1025	68/1/2	1270	63/1/1	1025	67/1/2
gcc	329	70/3/6	293	70/3/6	629	83/3/6	518	82/3/6
Mylyn	560	98/30/46	556	98/30/46	560	98/30/46	556	98/30/46
Bugzilla	379	100/6/11	342	100/6/11	406	100/6/11	374	100/6/11

Table 5.7: Configuring an assignment recommender with different heuristics and thresholds using the proportional data selection strategy.

Heuristics	Top 5				Top 10			
	Data Size	Sug. (Prec./Rec.)	Data Size	User (Prec./Rec.)	Data Size	Sug. (Prec./Rec.)	Data Size	User (Prec./Rec.)
Eclipse	1665	63/10/17	1320	57/9/16	2815	72/13/22	2280	67/12/20
Firefox	658	57/1/2	509	53/1/2	1128	65/1/2	829	65/1/2
gcc	329	69/3/6	286	72/4/8	619	80/3/6	484	82/3/6
Mylyn	508	97/30/46	199	97/30/46	508	98/30/46	199	98/30/46
Bugzilla	370	100/6/11	303	100/6/11	397	100/6/11	326	99/6/11

5.2.5 Summary of Results

These results show that a R_A with good precision can be created using a subset of the data and that the assisted configuration approach is tolerant to the use of a reasonable number of heuristics and threshold settings.

5.3 Configuring to Create Other Recommenders

Our assisted configuration approach is more general than the R_A case we have used as a running example. To discuss this generality, we describe how the assisted configuration techniques can be used to configure two other recommenders for bug reports, R_C and R_I.

5.3.1 Configuring a Component Recommender

In an issue tracking system, reports are often grouped by the functionality that they involve, called components. A common occurrence with open bug repositories[8] is that reports are filed under a default component, such as UI, as the reporter is often unable to determine a more suitable component. A component recommender can help by suggesting a more appropriate component based on the problem described in the report. Two modifications are needed to the assisted configuration process we have described in this paper to configure a component recommender.

First, it is no longer necessary to specify the data source for a path group because only one field is possible, the component field. Second, the process needs to help a user determine which components to recommend, rather than developers. Similar to the R_A example, the user could be presented with data about how many reports are filed under each component over a time period and a threshold suggestion.

5.3.2 Configuring an Interest Recommender

It is common that a bug report will have a list of individuals who want to be notified when a change is made to the report. For instance, if the bug report represents a problem, the individuals may be encountering the problem and want to know when the problem is fixed. Or, as another example, an individual may be interested because he is working on a bug for which this bug is blocking his progress.

To recommend who should be notified for a report change, we need to make two modifications to our process. We need to change the data sources that need to be associated with heuristics: two possibilities are the comments (specifically the names of the people who have submitted comments) and the cc: list for a report. We also need to allow a user to adjust for noisiness in this data. Similar to determining developer contribution

[8]An open bug repository is a repository that allows anyone with a user name and password to submit new reports or comment on existing reports.

for bug assignment, the level of noise could be adjusted by presenting the user configuring the interest recommender with a graph showing the occurrence of individual's names from the chosen data source(s) and a threshold suggestion for selecting the set of names to recommended.

5.4 Summary

This chapter presented an approach to assist a user to create a R_{DO} in a client-side environment using R_A as an example. We presented techniques for guiding a user in answering the questions of how many reports to use, which reports to use, how to label the reports, and which are valid labels. The assisted configuration approach had three steps. The first step helped the user to decide which reports and how to label them, by presenting them with a list of regular expressions representing the various life-cycles of the reports in the repository. For groups of reports that match a particular life-cycle, the user specifies a heuristic. The second step assists the user in deciding the labels by presenting a frequency graph and suggesting a frequency threshold. The final step uses the information from the other two steps to determine how many reports to use. We found that using this approach produces a R_A that is comparable to those created for the our evaluation in Chapter 4.

Chapter 6

Discussion

This dissertation presents a machine learning approach (ML$_{Triage}$) for creating development-oriented decision recommenders (R_{DO}). This chapter discusses a number of additional aspects about ML$_{Triage}$.

One aspect is the complexity of ML$_{Triage}$. Using ML$_{Triage}$ requires many inter-related decisions. Perhaps a more simpler approach for the different types of development-oriented decisions would be better? Section 6.1 explores this idea by comparing the use of a developer assignment recommender (R_A) created using ML$_{Triage}$ to a recommender that suggests the most active developers.

Another aspect of ML$_{Triage}$ is that it only takes into consideration one piece of process information: the categories for the development-oriented decisions. For example, when creating a R_A, ML$_{Triage}$ only considers how reports were assigned to the developers and does not consider other information such as that the developers of the project are divided into teams based on the product component. How can ML$_{Triage}$ use additional development process information to improve the accuracy of the created recommenders? Section 6.2.1 discuss the use of additional process information to improve the accuracy of a R_A.

This dissertation demonstrated how to use ML$_{Triage}$ to create three types of R_{DO}: a developer assignment recommender (R_A), a component recommender (R_C), and an interest recommender (R_I). How could ML$_{Triage}$ be used to create other types of R_{DO}? Section 6.3 explains how to use ML$_{Triage}$ to create three other types of R_{DO}. Similarly, Section 6.4 explains how the assisted configuration approach can be used for assisting in the configuration of recommenders that use source repository data.

In developing ML$_{Triage}$, we envisioned that the created recommenders would be used in a semi-automated fashion; the triager selects from a set of recommendations rather than a tool automatically applying a recommendation. However, could the recommenders created using ML$_{Triage}$ be used in an automated fashion or as part of the bug report submission process? Section 6.5 discusses how R_{DO} recommenders could be used to automate parts of the triage process or as part of the report submission process.

The ML_{Triage} process is batch-oriented: given a set of training reports, the process can produce a recommender based on those training reports. However, in practice, bug reports arrive and are updated continuously in the repository. We describe how ML_{Triage} might be better adapted to this reality by considering a machine learning algorithm that updates the recommender's model on-the-fly as data flows into the repository. Finally, we discuss one of the drawbacks to the ML_{Triage} approach: not taking into consideration direct feedback from the triager.

6.1 Using a Naïve Approach to Development-Oriented Recommender Creation

Creating recommenders using ML_{Triage} is a complex process as there are many inter-related decisions that need to be made. For example, choosing how to label the reports can affect which labels are determined to be valid. However, is such complexity necessary?

A simpler approach than ML_{Triage} could be to recommend the most frequently occurring category values. This approach might be feasible because the number of reports within different development-oriented categories are commonly imbalanced. For example, there is commonly a small core set of developers who resolve most of the reports [25, 46]. Another example, is product components under which most reports fall such as General for Firefox and UI for Eclipse. One approach to creating R_{DO} recommenders could be to suggest the most frequently occurring categories (e.g., the top four fixers or the top three components). We refer to this approach as the *naïve approach* to R_{DO} creation.

Table 6.1 compares the precision of the naïve approach with that of ML_{Triage} for a developer recommender (R_A). We present results for each of the five projects that we used for evaluation in Chapter 4. In the case of Eclipse, the naïve approach creates a very poor recommender as the precision for one recommendation is 23% compared to 75% for the recommender created using the ML_{Triage} approach. For Firefox and Bugzilla, the ML_{Triage} recommender appears to be slightly better than the naïve approach. The ML_{Triage} recommender for Mylyn is similarly comparable to the naïve approach for the one and two recommendations, but the precision drops significantly for three recommendations. Finally, the naïve approach for gcc is comparable to ML_{Triage} for one recommendation, but has lower precision for two and three recommendations.

For analysis, we performed paired t-tests (i.e., repeated measures) between the precisions[1] calculated using the Support Vector Machines and naïve approaches across the five projects; a separate paired t-test was performed for each of the three sizes of recommendation lists. Table 6.2 shows the results of the statistical tests. We found that for all three prediction list sizes there was no statistically significant difference between the two types

[1]The precisions used in this calculation are the average precisions across the test cases.

Table 6.1: Precision of naïve approach to the ML$_{Triage}$ approach for creating a R_A (SVM=Support Vector Machines NA=Naïve approach).

	Eclipse		Firefox		gcc		Mylyn		Bugzilla	
Predictions	SVM	NA	SVM	NA	SVM	NA	SVM	NA	SVM	NA
1	75%	23%	70%	70%	84%	84%	98%	100%	98%	96%
2	60%	26%	65%	66%	82%	55%	93%	94%	98%	98%
3	51%	23%	60%	57%	76%	62%	82%	63%	92%	93%

Table 6.2: Results of a paired t-test comparing the two types of recommenders.

	t	df	p
1	1.00	4	0.375
2	1.53	4	0.201
3	2.39	4	0.075

of recommenders at our chosen 95% confidence level.

We are confident in the values for the Eclipse project due to our work on estimating implementation expertise sets [5]. For the Mylyn and Bugzilla projects, we believe that both of these projects have a small number of developers, so a naïve approach will work well. Finally, we believe that the results for the Firefox and gcc projects are the result of over estimating the implementation expertise lists, however it is difficult to tell if this is the case.

Another consideration to be taken into account in selecting between the approaches for a particular project is the nature of the two approaches. Ignoring the actual results, by its nature the naïve approach will always recommend the developers that are contributing the most to the project, regardless of their specific expertise. This means that if the top recommendations are always accepted for the naïve approach then all the work will be directed to those who are doing the most work on the project. This will obviously lead to a work imbalance across the project developers. For a project with few developers this result may be reasonable, but for a project with many developers this result is likely undesirable. As the ML$_{Triage}$ approach bases recommendation on evidence of expertise, not on activity, a recommender created using ML$_{Triage}$ is less likely to have this problem, especially as the code base grows and developers can no longer be general experts.

6.2 Using Other Development Process Information to Create a Development-Oriented Decision Recommender

The ML$_{Triage}$ approach to creating a recommender only takes into consideration one type of information, namely the category for which the recommender is being created. For example, in creating a R_A, ML$_{Triage}$ only considers information about the individual developers. Could a more accurate R_{DO} be created if we took into consideration more information about the development process? For example, could a more accurate R_A be created if we incorporate knowledge such as that developers are structured into teams based on the product component? This section explores the idea of using more development process information to create a R_{DO} called a *component-based developer recommender.*

6.2.1 Component-based Developer Recommendation

We observed that the development teams for the Eclipse and Firefox projects were formally structured around the project's components. This structuring of development teams was confirmed through communication with these two projects and has also been observed with other projects [23, 46]. This structuring can be an important consideration for triagers when doing developer assignment. As mentioned in Section 2.1.1, the Eclipse project first triages reports according to the component that the report is against and then reports are triaged by the component team. A potential way to improve the developer recommender precision is to incorporate this process information into $\mathrm{ML}_{Triage}$and create a component-based developer recommender (R_A^C).

To construct a R_A^C, we follow a process analogous to that used to triage Eclipse reports (see Section 2.1.1). The data is first grouped by reported component and then the groups of reports are used to train a separate recommender for each component. As before, a developer profile of an average of three resolutions for each of the past three months is used to determine which developers to recommend. The profiles are created without regard to the components as we had found that developers often work across multiple components [4]. In other words, even though developers are formally divided into teams based on component, they do not strictly develop within that component. The work of Minto and Murphy described this phenomenon as *emergent teams* [44]. For a new report, the recommender uses the appropriate component-recommender based on the value of the `component` field in the report to make the recommendations.

Table 6.3 shows the analytical results of using this process for the five projects to create a R_A^C. The results of the analytical evaluation of R_A from Chapter 4 are reiterated in Table 6.4 for comparison. From Tables 6.3 and 6.4 we see that using the process information substantially improved the precision of the R_A^C recommender for Eclipse, but generally degraded the precision of the other recommenders. This result may be a consequence of Eclipse developers working more strictly within the component boundaries than the developers of the other projects [4], however data from other work seems to indicate that this is not the case [44]. This result may also be a consequence of emergent teams being more prevalent for the other projects.

Table 6.3: Precision and recall for a component-based developer recommender (R_A^C) for the five projects.

Predictions	Eclipse (P/R/F)	Firefox (P/R/F)	gcc (P/R/F)	Mylyn (P/R/F)	Bugzilla (P/R/F)
1	97/18/30	70/1/2	92/4/8	94/29/44	94/5/9
2	93/34/50	64/2/4	79/6/11	90/53/68	94/11/20
3	79/41/54	64/3/6	78/10/18	82/73/77	82/12/21

Table 6.4: Precision and recall for a developer recommender (R_A) for the five projects.

Predictions	Eclipse (P/R/F)	Firefox (P/R/F)	gcc (P/R/F)	Mylyn (P/R/F)	Bugzilla (P/R/F)
1	75/13/22	70/1/2	84/3/6	98/30/46	98/5/10
2	60/20/30	65/2/4	82/6/11	93/55/69	98/11/20
3	51/24/33	60/3/6	76/10/18	82/72/77	92/14/24

6.3 Creating Recommenders for Other Development-Oriented Decisions

This dissertation has shown how ML_{Triage} has been used to create R_{DO} recommenders for three decisions made by triagers. This section presents an overview of how ML_{Triage} could be used to create three other types of R_{DO} recommenders. The first recommender is a triage recommender that recommends the sub-component against which to file a report. The second recommender is a recommender that estimates the effort required to fix a report. This recommender could also be used by developers or project managers to prioritize work. The final recommender is an impact analysis recommender which suggests the files that will be affected by the fix of a report. This last recommender could be used by both a project manager or a developer.

The sub-component recommender was implemented as part of the field study and we present both an analytical and field evaluation of the recommender as was done for the R_A, R_C, and R_I recommenders presented in Chapter 4. For the other two recommenders, the effort estimation recommender and the impact analysis recommender, we only present an overview of how ML_{Triage} could be used to create the recommenders. The implementation and evaluation of these two recommenders is left to future work.

6.3.1 Sub-component Recommender

During the field study, the Eclipse triagers requested that a sub-component recommender (R_S) be created using ML_{Triage}. They made this request because the component for which they triaged, UI, was further divided into sub-components and developer assignment decisions were based on the sub-component. This section presents the results of an analytical analysis of the sub-component recommender and results from the use of the sub-component during the field study.

Table 6.5: The sub-components for the Eclipse project.

Component	sub-components	Training Reports
CVS	12	174
Debug	6	176
Runtime	4	33
SWT	4	35
Team	11	197
Text	15	266
User Assistance	6	261
UI	56	1850

Table 6.6: Answers to the six questions for ML_{Triage} when creating a sub-component recommender.

Question	Answer
Which reports?	Assigned and resolved reports are used and partitioned based on the value of the `component` field.
How many reports?	Reports are collected from an eight-month period.
Which features?	The summary and description are used for the features.
How to label?	Reports are labeled by the sub-component information appearing in the report summary.
Which labels are valid?	All sub-component labels are considered valid.
Which algorithm?	The Support Vector Machines algorithm is used to create the recommender.

Creating a Sub-component Recommender

Of the eighteen components of the Eclipse project, eight components had sub-components. Table 6.5 shows the number of sub-components for these components and the number of training reports for all the sub-components of the component. From the table we see that the number of sub-components ranges from four (SWT) to fifty-six (UI).

A R_S is created in a similar way to that of a R_A^C (see Section 6.2.1). The recommender is created by dividing the reports by the value of the `component` field and creating a recommender for each report group. Table 6.3.1 shows the answers to the six questions from ML_{Triage} for creating a R_S with the differences in answers between a R_A and a R_S highlighted.

The process for obtaining the sub-component label requires explanation. As Bugzilla does not have a predefined field for sub-components, the Eclipse project convention is to specify the sub-component in the summary by enclosing it in square braces. For example, a report summary starting with "[Workbench] ..." refers to a bug report for the "Workbench" sub-component (see Figure 6.1) Reports are therefore labeled with their sub-component using the heuristic of extracting the words contained in the first set of square braces found in the summary field.

Figure 6.1: Example of an Eclipse report for the sub-component 'Workbench'.

Table 6.7: Recall for a sub-component recommender.

Recommendations	Recall
1	49%
2	54%
3	57%

Analytical Results

Recall from Section 4.1.3 that a report will only belong to one project component. Similarly, a report will only belong to one sub-component. For the same reason that precision does not provide an appropriate evaluation of a R_C, we only present the recall for evaluating a R_S. The training reports were gathered from the resolved reports that could be labeled with a sub-component from the time frame of October 2005 to May 2006. Our testing set contained 877 reports, which were the assigned and resolved reports of the Eclipse project from June 2006 that had a sub-component specified.

Table 6.7 shows the analytical results for sub-component recommendation for the Eclipse project. With three recommendations, we achieve a recall of 57%. This result means that with three recommendations, the correct sub-component was recommended just over half of the time. Given that the sub-components of the project represent a fine granularity of project functionality, and therefore there is relatively little training data for each sub-component compared to that used for the other recommenders, this result is not surprising.

Field Study Results

Quantitative Results During the field study (see Section 4.2), we recorded 188 sub-component assignments. As mentioned in Section 3.5, Sibyl provided three sub-component recommendations. We found that in practice, R_S provided the correct sub-component recommendation just under half (48%) of the time. Considering that the sub-component labels are less constrained than the component labels (i.e., it is very easy to add a sub-component ad-hoc), and compared to the other recommenders the classes for the sub-component recommender had the least amount of data, it is not surprising that R_S did not make very accurate recommendations.

If the cases where the sub-component recommender could not reasonably provide correct recommendations are removed from consideration (i.e., the twelve cases where the report had been filed under the wrong component), then the accuracy improves marginally to 52%. This refined accuracy approaches the 57% accuracy we determined analytically.

As was mentioned in Section 4.2, as we did not have information about what was the correct number of sub-component recommendations to make, we chose to provide three recommendations, as this was the same number of recommendations we provided for the R_C. However, as with the other recommenders in Sibyl, we logged more recommendations than were presented to the triager. As with R_C we recorded six recommendations in the log. We found that if the number of sub-components recommendations presented had been raised from three to six, that the R_S would have an improved in-practice accuracy and refined accuracy of 55% and 59% respectively. This result shows that even with doubling the number of recommendations, the accuracy of the sub-component recommender would only improve by less than ten percent.

Qualitative Results As was done for R_A and R_C, a decision questionnaire was given to the triagers for the sub-component recommender.

For sub-component recommendation, five responses indicated the triagers felt that three recommendations was a reasonable number of recommendations. However, three responses indicated that three recommendations was too few. We believe this result is likely due to the low accuracy of the recommender; the triagers wanted to find the right recommendation in the list and believed that it was lower down. As with the component recommender, the triagers indicated twice that there were multiple applicable recommendations, and that they felt that it was easy to decide between them. In the usage questionnaire filled out by one of the triagers, the triager commented that she found R_S to be useful.

6.3.2 Effort Estimation Recommender

Some projects, such as JBoss[2], track the effort required to fix bugs and add features in the bug repository.[3] Previous work by Weiß and colleagues presented a technique for estimating how long it will take to fix a bug (a development-oriented decision) based on effort information found in the bug reports [65]. To recommend how long it will take to fix a new report, they use the nearest-neighbour algorithm and compare the summary and description of the bug reports. Using their approach only 30% of their predictions were within a $\pm 50\%$ range of actual reported effort.

In effect, Weiß et al. used a rougher version of ML_{Triage}.[4] To use ML_{Triage} to create a recommender that estimates effort (R_E), the six questions would be answered as shown

[2]JBoss is an Java application server; available at `labs.jboss.com/jbossas`, verified 14/08/07

[3]None of the projects that we examined tracked bug fixing effort.

[4]The authors acknowledge that their work was motivated by our ML_{Triage} work.

in Table 6.3.2, with the differences between answers for a R_A and a R_E highlighted. The creation and analysis of this type of R_{DO} is left as future work.

Table 6.8: Answers to the six questions for ML_{Triage} when creating an effort estimation recommender.

Question	Answer
Which reports?	Resolved reports are used.
How many reports?	Reports are collected from an eight-month period.
Which features?	The summary and description are used for the features.
How to label?	Reports are labeled by effort bin. The effort values (e.g., 1.5 hours) would be discretized into bins such as < 1 hour, 1-2 hours, or 1 day.
Which labels are valid?	All effort ranges would be considered valid.
Which algorithm?	The Support Vector Machines algorithm is used to create the recommender.

6.3.3 Impact Analysis Recommender

A common development-oriented decision is *impact analysis* during which the set of source files that need to be accessed to fix a fault or implement a feature are determined [48, 70, 71]. Canfora and Cerulo used an information retrieval technique for doing impact analysis of bug reports [13]. In their approach, they extracted text from both past reports and source repository logs. After processing and indexing the terms, their system returned a list of possible files to change based the text of a new report. With their approach they were able to predict impacted source files with a top precision and recall of 36% and 67% respectively for Firefox.

By linking a bug report to the source files impacted by the report, ML_{Triage} could create a source file recommender (R_F). Table 6.3.3 shows the answers to the six ML_{Triage} questions for such a recommender with the differences between answers for a R_A and a R_F highlighted. The creation and analysis of this type of R_{DO} is left as future work.

6.4 Assisted Configuration of Recommenders for Other Repository Types

We believe that elements of our assisted configuration approach can also apply to recommenders built for other kinds of repository artifacts. As one example, both Zimmermann and colleagues [71] and Ying and colleagues [70] have proposed approaches that, given a set of source files a developer is editing, recommend other files that the developer should consider changing based on co-occurrences of changes to these files recorded in the source repository. These kinds of approaches also require configuration, such as determining which transactions to the source repository to consider when marking co-occurring changes. Sim-

Table 6.9: Answers to the six questions for ML_{Triage} when creating an impact analysis recommender.

Question	Answer
Which reports?	Resolved reports are used.
How many reports?	Reports are collected from an eight-month period.
Which features?	The summary and description are used for the features.
How to label?	Reports are labeled by the names of files that were touched by the fix.
Which labels are valid?	All file names could be considered valid. An alternative is to use degree of interest information, such as Mylyn collects, to restrict the labels to the names of files that were most important for the fix [35].
Which algorithm?	The Support Vector Machines algorithm is used to create the recommender.

ilar to the approach used for determining developer contribution levels for a R_A (see Section 5.1.2), a distribution graph of the size of transactions could be presented to the user to aid in configuring a file recommender. These approaches could also be extended to have the user configure which kinds of transactions to consider, similar to the path group selection for R_A configuration (see Section 5.1.1). As one example, the transactions could be grouped, perhaps based on whether the transactions solved similar problems based on bug similarity, and characterized, perhaps through keyword extraction, and presented to the user for selection.

6.5 Using Recommenders to Automate Triage Decisions

In developing ML_{Triage}, we envisioned that the created recommenders would be used in a semi-automated fashion; the triager selects from a set of recommendations rather than a tool automatically applying a recommendation. Our rationale is that in making a triage decision, the triagers draws on knowledge that is not necessarily available to the recommender. For example, when using a recommender to suggest to whom the report should be assigned for resolution, the triager might choose from the set of developers the tool recommends based on such knowledge as the current workload of each recommended developer, who is on vacation, or other information that was not available in the bug repository and thus not available to the recommender.

However, we found that for some projects, ML_{Triage} can create R_{DO} which have a high enough precision or recall that it may be possible to use the recommenders to either automate certain development-oriented decisions or as part of the report submission process. For example, the R_A for the Mylyn and Bugzilla projects had precisions above 90%. This means that reports for these projects could be assigned automatically with few errors. We also found that the R_C for all five projects were above 75% for three recommendations. This presents the possibility of having the submitter of the bug report select from the product component recommendations instead of the longer list presented by the issue tracking system. For projects, such as Eclipse Platform, where reports are triaged within the team for a particular report, providing the R_C to the submitter may reduce the number of misclassified reports that the triagers have to examine. However, more study of the impact of R_{DO} recommenders on the triage process is needed.

6.6 Using an Incrementally Updated Algorithm

The approach we present in this dissertation trains the classifier using a batched set of data; all the data is gathered together and fed to the machine learning algorithm so that the recommender has a complete view of all the training data.. In contrast, an incrementally updated algorithm only views the instances of the training data one at a time and the classifier model is updated accordingly [60].

There are two potential advantages to using an incrementally updatable algorithm. The first is that using an incrementally updated algorithm mimics how information flows in to and alters a bug repository.

The second is that using an incrementally updated algorithm allows the classifier model to be created on the fly. As explained in Section 3.5, the Sibyl classifiers are updated by periodically downloading new training reports and adding them to the set of reports to be used for training. When the recommender is trained, all the data is presented to the algorithm at the same time. The process for updating client-side recommenders would be the same; download the new training reports and present all the training reports to algorithm in a batch. The new classifier is trained offline and then made active when training has completed. A classifier trained with an incrementally updated machine learning algorithm does not need to use an offline training process. As with updating the batch-trained recommenders, new reports would be downloaded periodically, however the reports could be integrated into the recommender as they are accessed. This may result in the classifier being more responsive to project changes such as the addition of new developers or product components.

Figure 6.2 and Figure 6.3 provides a view of the performance of an incrementally updated Naïve Bayes algorithm to create a R_A where the reports are added to the recommender in chronological order. We chose to use the Naïve Bayes algorithm for our comparison as we did not have access to a incrementally updatable version of Support Vector Machines. The recommenders are evaluated after fifty new reports are added to the recommender. Figure 6.2 compares the precision of the incrementally updated classifier for Eclipse and Firefox. The precision and recall of the Eclipse and Firefox R_A created using Naïve Bayes (see Chapter 3) have been added to the figures for comparison. As the precision for the batch-data Naïve Bayes R_A would appear as a dot in the graph, we have

Table 6.10: Precision, recall and F-measure when using an incrementally updated Naïve Bayes algorithm.

Predictions	Firefox	Eclipse
	(P/R/F)	(P/R/F)
1	50/0.5/1	5/0.6/1
2	49/1/2	3/0.6/1
3	47/2/4	3/1/2

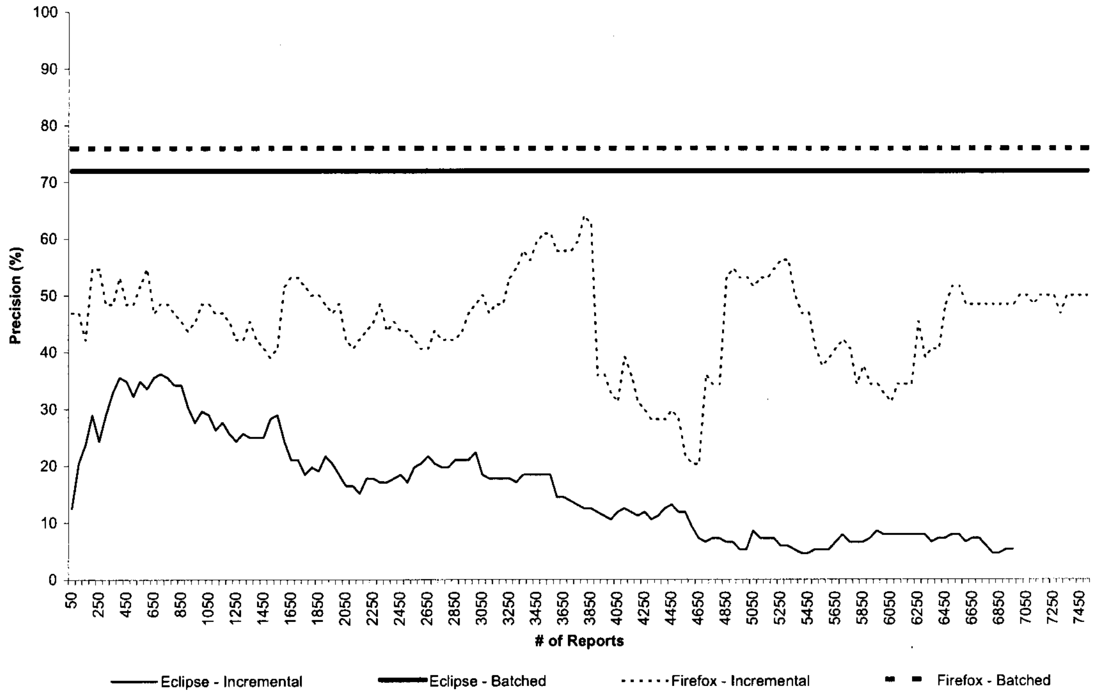


Figure 6.2: Precision of a developer classifier created using an Naïve Bayes algorithm that is incrementally updated.

interpolated the value. Similarly, Figure 6.3 shows the recall of the incrementally updated classifiers for the two projects, with the recalls for the batched-data Naïve Bayes R_A also interpolated. We used the same data and data preparation process as in Section 3.2.6.

From the two figures we see that using an incremental Naïve Bayes algorithm does not produce very good recommenders in the best case (i.e., using only the most recent 700 reports from Eclipse or 3800 for Firefox) as the best precision reached for both projects is below that achieved when giving all of the reports to a Naïve Bayes algorithm in a single batch. In each case, the recommender's precision and recall degrades substantially over time.

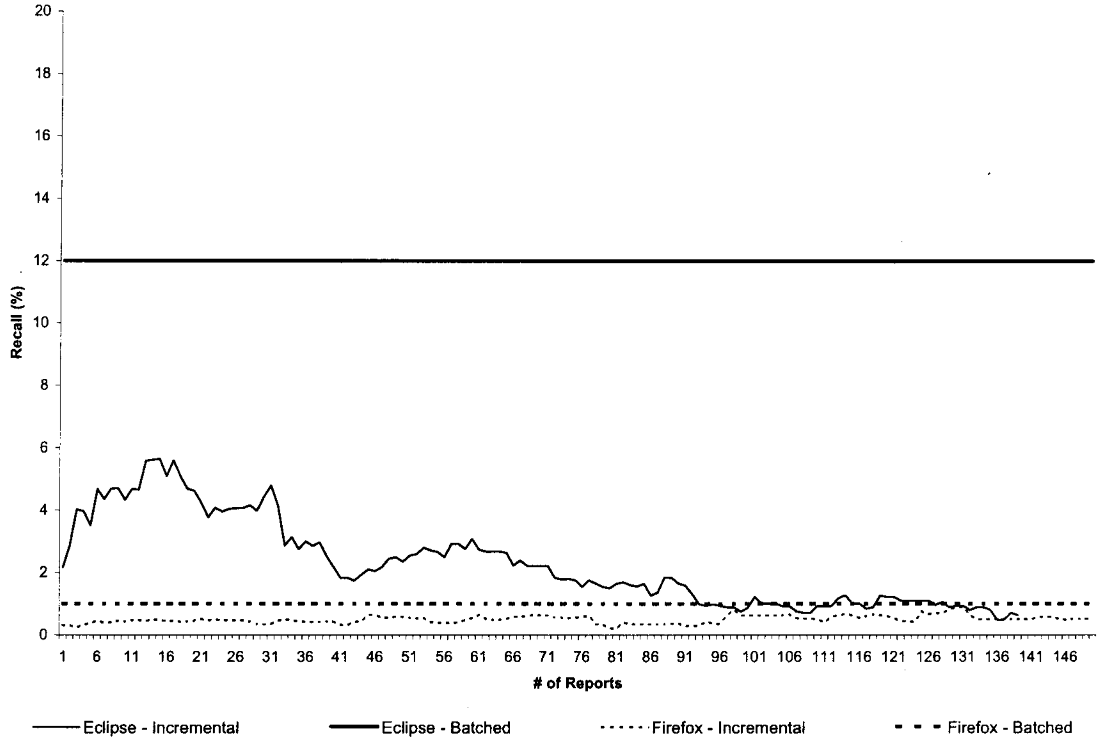


Figure 6.3: Recall of a developer classifier created using an Naïve Bayes algorithm that is incrementally updated.

6.7 Providing Feedback to the Recommender

A limitation of the recommenders created using ML$_{Triage}$ is that they do not incorporate direct feedback from the triager about the recommendations. For example, a component recommender R_C could be created using ML$_{Triage}$ and then used by a tool that presents the triager with the top recommendation. If the recommendation is not appropriate (i.e., the triager does not like the recommendation), the triager could ask for a new recommendation and the tool would move down the list. However, in not accepting the first recommendation, the triager has provided some feedback; they have provided a negative case that might be used to improve future recommendations. However, the algorithms that we investigate do not directly take such feedback into account; the algorithms use positive labeling information not negative labeling information. In other words, the algorithms require the label to state "This instance belongs to this class" and do not directly use the information in the statement "This instance does not belong to this class".

Prior work examined the used the use of feedback for a R_C [23]. Di Lucca and colleague's approach to incorporating feedback in a component recommendation system was to present the next likely component if the triager did not like the recommendation. However, we disagree that this is truly providing feedback to the recommender, as the recommender is not incorporating any new information from the rejection of the recommendation; if the recommender was immediately presented with the same report it would produce the same set of recommendations.

A potential technique for incorporating feedback into the recommender is to use reinforcement learning [45, 64]. Reinforcement learning is concerned with how an agent should behave so as to maximize a long-term reward. Reinforcement learning tries to determine a policy that maps the states in the world to how the agent should react to the states. In the context of R_{DO}, the recommender would make a recommendation and would receive a positive or negative answer. Based on that answer the reçommender would make new recommendations based on trying to maximize the reward (i.e., lower the recommendation error rate). However, as a reinforcement learning algorithm is designed to trade-off long-term and short-term reward, we believe that a triager or developer would likely give up using the recommender before the recommender achieved a sufficient level of accuracy.

Chapter 7

Related Work

In this chapter we present work related to our machine learning-based approach for creating development-decision recommenders, ML_{Triage}. We begin by presenting an overview of other work that has observed the triage process and how reports are assigned to developers, one of the development-oriented decisions. Next, we discuss various work that has mined data from software repositories. Since one of the types of development-oriented decision recommenders, the developer assignment recommender, is a form of expert recommender, we present an overview of various work that has looked at assisting with expert recommendation, both inside and outside the field of software engineering. We end this chapter with comparisons of other work that looks at providing support for triagers through recommendation.

7.1 Studies of Developers doing Triage

To the best of our knowledge there have not been studies that specifically look at the process of triage. However, other studies of developers do provide glimpses of the triage process, primarily how developer assignment is done.

Sandusky and colleagues examined 182,000 reports from a large, thriving open source development community to investigate the principle of negotiation in the coordination of distributed software problem management [57]. They identified triage and assignment as one of the six steps in the "normative software problem management process" [57, page 189] and one of the two steps on which they focused to investigate negotiation. Of the thirteen topics found to be negotiated, "who is responsible for fixing the bug?" was the sixth highest. Of the five topics that were higher in rank, three were repository-oriented decisions ("is there a problem?', "is this a duplicate [report]", and "is this problem fixed?'). The remaining two were development-oriented decisions that can not be categorized ("what is the best design?" and "how should the [report] be managed?").

Crowston observed the software change process used in the minicomputer division of a large corporation [16]. He used a coordination theory approach to present two alternatives to their report assignment process. The corporation's existing assignment strategy was to assign reports to the developer in charge of the module that appeared in error (called the specialist strategy). Crowston discussed the advantages and disadvantages of using a generalist strategy whereby reports were given the next available developer and a market strategy in which developers bid on the reports for which they will assume responsibility. As ML_{Triage} builds it recommenders based on historical evidence, which neither the general or market use, it is inherently a specialist strategy.

Mockus and colleagues examined the development process used by two open-source software projects, Apache[1] and Mozilla, and compared them to five commercial projects [46]. For the five commercial projects, they reported that reports were assigned to developers by a supervisor according to developer availability and type of expertise required. A developer recommender created using ML_{Triage} assists the supervisor with the latter. For both the Apache and Mozilla projects, it was reported that developers were allowed to self-assign reports based on their code ownership, expertise, and interest. Mockus et al. also reported that for the Apache project, triage was done by one or two interested developers who performed periodic triage of new requests. From our observations this is consistent with other open-source projects; although there may be various volunteers that help with triage, there is usually one or two key people performing triage.

Ko and colleagues observed seventeen Microsoft developers to determine their information needs [39]. They found that bug triage was a common developer activity, observing that 9 out of the 17 developers triaged reports during the observed 90 minute period in which they were observed. They found that the primary information needed by the observed developers when doing triage was to determine if the problem was legitimate (a repository-oriented decision), how difficult will it be to fix the problem (a development-oriented decision) and is the problem worth fixing (a repository-oriented decision). ML_{Triage} could provide a recommender that would assist the developer in estimating the effort needed to fix the report if the reports in the project's repository had information about the effort required to fix reports (i.e., how difficult to fix) and the effort values were discretized (e.g. < 30 minutes, < 1 hour, or < 5 hours). See Section 6.3 for more details.

Carstensen and colleagues analyzed the coordination work of software testing at a Danish manufacturing company to promote general requirements for computer support [15]. They observed that triage, including the assignment of reports to developers, was handled in weekly meetings. They suggested that a better classification system for the type and importance of bugs would have facilitated the triage process. From informal conversations with developers working on commercial projects, this assignment technique is still used, but less often due to improved coordination tools such as on-line bug repositories.

[1]Apache is a popular web sever; `http://httpd.apache.org`, verified 14/08/07.

7.2 Mining Software Artifact Repositories

Recent years have seen an increase in the mining of repository data from software projects, primarily the source repository, bug repository, and email archives. We divide these efforts into categories based on the type of repository that is mined.

7.2.1 Mining Source Code Repositories

Source repositories have been mined for a variety of purposes including predicting faulty modules [9, 37, 49], recommending other files to change [70, 71], and code ownership [8]. We highlight efforts from each of these topics.

Brun and colleagues proposed a program analysis technique for determining which program properties may indicate faults [9]. In their approach, a program analysis tool generates program properties for faulty and non-faulty versions of the the programs. These properties are then given to a machine learning algorithm, and models of these properties are created. Brun et al. experimented with using the Support Vector Machines and C5.0 (a revised version of C4.5) to create these models. Based on experimentation, they found that the Support Vector Machines algorithm was very good at ranking the program properties to indicate which properties most indicated faulty software.

Both Ying and colleagues [70] and Zimmermann and colleagues [71] proposed approaches that, given a set of source files a developer is editing, recommend other files that the developer should consider changing based on co-occurrences of changes to these files recorded in the source repository. Their approaches were based on data mining techniques and were found to discover hidden program dependencies. For example, on checking in a set of files to the source repository, the developer could be warned about missing files (i.e., files that should have also been changed).

Bowman and Holt mined source repositories to determine the *ownership architecture* of a system [8]. The ownership architecture shows how developers are grouped into teams and relates these teams to the code for which they are responsible. Proposed uses of an ownership architecture included the identification of experts, the identification of non-functional dependencies, software quality estimates, code abandonment, ownership coverage, and overstaffing or understaffing of subsystems. However, the approach was only manually applied to the Linux project as a proof of concept, and never employed in practice.

7.2.2 Mining Bug Repositories

Information stored in a bug repository has been used by a number of researchers. Some have used the information to investigate questions about the development processes [57, 36], some about relationships between artifacts of the development process [28, 66] and others for providing triagers and project managers with recommendations [55, 32]. We highlight efforts from each of these categories that have relevance to our work.

Mockus, Fielding and Herbsleb used data from the source and bug repositories of two open source projects —Mozilla and Apache— to compare the open source software development (OSS) process to traditional commercial development methods [46]. Using the bug repository, they arrived at conclusions about such things as the roles played in the OSS development community, distribution of work, and the defect densities of the two products. One finding was that OSS projects tend to have a group one order of magnitude greater than a core developer group who repair defects and receive bug reports from a group two orders of magnitude greater than the size of the core development team. This one to two orders of magnitude difference between the group reporting the bugs and the group fixing the bugs makes triage a problem for such projects.

Work by Sandusky and colleagues focus on the information found in the bug repository. The goal of this work was to identify *bug report networks* [58]. Bug report networks are groupings of bug reports due to duplication, dependency or reference relationships. These relationships are described in the reports as a means of improving how problems are managed. This work relates to ours on two levels. At a low level both our work and the work of Sandusky et al. investigate the use of grouping reports by a common feature. In their work it is the relationships that are explicitly stated in the report. In our work, the grouping is by development-decision categories. At a higher level, both works aim to address the management of bug reports. In the case of Sandusky and colleagues, bug report networks have two purposes: as an information structuring strategy to reduce cognitive and organization effort and as a social ordering mechanism of different community roles such as reporter, assigned-to, or cc: list member. Our work looks at assisting with the management of reports by providing categorization recommendations for new reports.

Weiß and colleagues developed a technique for estimating how long it will take to fix a bug (a development-oriented decision) based on effort information found in the bug reports [65]. To recommend how long it will take to fix a new report, they use the nearest-neighbour algorithm and compare the title and description of the bug reports. Using their approach only 30% of their predictions were within a $\pm 50\%$ range of actual effort. In effect, Weiß and colleagues used a rougher version of ML_{Triage}.[2]

7.2.3 Email Archives

Bird and colleagues mined the email archive of the Apache HTTP server project to answer questions about the communication and coordination practices of developers in open source software [7]. They examined questions about the properties of developer social networks, correlation between activity on mailing lists and activity in the source repository, and developer status. Bird et al. found that the email social network was typical of other electronic communities, such as a few members accounting for most of the messages. They

[2]The authors acknowledge that their work was motivated by our ML_{Triage} work.

found that there was a strong correlation between mailing list activity and development activity and developers do play a more significant social role than all the participants on the mailing list.

Just as Bird et al. observed that a few members account for many of the email messages in the mailing list, we observed that bug reports tend to be unevenly spread across the different development-oriented decision categories. This uneven distribution across the categories leads to the need to specify which labels are valid when creating a R_{DO}.

7.3 Expertise Recommendation

One kind of recommender created using ML_{Triage} is an assignment recommender that suggests which developer to assign a report, R_A. This kind of recommender is an *expert recommender.* In this section, we discuss other work that has examined the recommendation of experts, both inside and outside of the software engineering field.

7.3.1 Expertise Recommendation in General

Outside of the field of software engineering there have been efforts using a variety of techniques for recommending experts. In this section we discuss expert recommenders that use matrices, graphs and information retrieval. Further examples of automatic expert finders are listed by Yimam-seid and Kobsa [69].

One technique used for expert recommendation is the use of expertise matrices. Streeter and Lochbaum presented a technique that used a matrix to categorize different departments of a research and development company [63]. Five hundred departments were categorized based on the titles of the documents produced by the departments. One hundred areas of knowledge, or concepts, were determined using *latent semantic analysis* [21] and used as the rows of an expertise matrix. An individual would query an expert recommender tool and the terms of the query were used to extract appropriate rows from the expertise matrix. The values of the rows were then used to provide a ranked list of departments.

Another technique used for expert recommendations is expert graphs. Campbell and colleagues built expertise graphs for the topics appearing in email messages [11]. Emails submitted by users to the expert recommender system were clustered based on text and the relationships between the senders and receivers of messages in the clusters was created. A graph-based ranking algorithm (HITS [38]) was then used to determine the user's topic expertise.

Another graph-based approach is the *ExpertiseNet*, which creates expertise graphs of researchers based on their publications [62]. ExpertiseNet uses the text and citations of publications to create an expertise graph of a researcher. The citations are used to gauge how an author influences certain fields based on the fields that her papers cite and the fields that cite her paper. Latent semantic analysis [21] is used on the text of the document to determine how related two documents are. An expert can be found by keyword searching on the expertise nets.

A final example of how a system can provide expert recommendations is by information retrieval. The *ExpertFinder* system developed for the MITRE corporation parsed documents on the corporate intranet, such as technical papers, presentation, resumes, announcements and newsletters. ExpertFinder then provided a search engine interface that returned expert recommendations based on frequency of terms associated with particular employees [41].

Our work uses a fourth technique, machine learning, for creating the expertise model of the developers. This technique is most similar to the information retrieval approach used by the ExpertFinder in that we parse text to determine expertise. However, a machine learning approach differs from an information retrieval approach in that we are not creating an index of terms that point to developers, but are instead building a more general model of developer expertise.

7.3.2 Expert Recommendation in Software Engineering

Expert recommendation in the software engineering domain has primarily focused on the use of the source code repository. We provide three such examples.

The *Expertise Recommender* by Macdonald and Ackerman [43] provided expertise recommendations for two groups, a technical support group and a software development group. The expertise was determined based on two heuristics. The heuristic for the technical support group was based on a keyword search applied to the titles of other problem reports. The heuristic for the development group was the Line-10 rule, which determines the user name of the developer who made changes to a file in the source repository. The expert for the file was deemed to be the developer who last checked in a change to the file.

Another system that used a similar approach is Mockus and Herbsleb's *Expertise Browser*. The Expertise Browser provided recommendations of developers who have expertise for source code files. They based the level of a developer's expertise on the number of expertise atoms the developer had for the file, where an expertise atom was a change to the file that was checked into the source repository. Like the Expertise Recommender, they used the Line-10 rule, however where the Expertise Recommender only used the top name, the Expertise Browser used all the names from the file's source repository log. The Expertise Browser contained an interface whereby a user selected a file and received a ranked list of experts for the file, with the amount of expertise being indicated by font size. The user could also select a developer and see which files they had worked on, and similarly see how much they had modified the file. The Expertise Browser was deployed in a large, distributed software development project, and found to be most useful in locations that were either new to the project, or lacked local system expertise.

The *Emergent Expertise Locator* (EEL) by Minto and Murphy also used the Line-10 approach to determine expertise for *emergent development teams* [44]. An emergent team is an ad-hoc team that spontaneously forms around solving particular problem. EEL provided recommendations for members of the emergent team based which files have changed together in the past and which developers made the changes. The EEL approach used matrix-based computation to provide the expertise recommendations for the set of files that the developer was currently interested in.

7.4 Assisting Triage Through Recommendation

There are three other works that have looked at creating instances of R_{DO} recommenders. Two have looked at creating developer recommenders (R_A) and one has looked at creating a component recommender (R_C). To our best knowledge, none of these works have been incorporated into a tool for use by triagers. There have also been efforts to assist with one of the repository-oriented decisions: detecting reports that describe problems already represented in the repository (i.e., duplicate reports).

Di Lucca and colleagues [23] investigated the use of different machine learning algorithms to recommend which of eight maintenance teams to which a report should be assigned. Although they referred to their recommendations as development team recommendations, as the development teams were each responsible for one product component, from our point of view this is the same as component recommendation. Similar to our work, they investigated the Support Vector Machines, Naïve Bayes, decision trees, and nearest neighbour algorithms, and found that both Naïve Bayes and Support Vector Machines were appropriate algorithms (see Section 3.2.6).

Čubranić and Murphy [19] presented an approach to creating a R_A based on text categorization. In their work they achieved precision levels of around 30% on data from the Eclipse project using the multinomial Naïve Bayes algorithm. Similar to our work, they used heuristics for determining the label of the training reports (see Section 3.2.4).

Canfora and Cerulo [12, 14] outline an approach for developer recommendation based on information retrieval. They used text from both the bug report description and source repository logs and prepared the text in a manner similar that for ML_{Triage}. Also, they use a technique analogous to our use of the Naïve Bayes algorithm for making the recommendations. Their approach achieved a precision of around 20% for the Mozilla project.

The work of both Runeson and colleagues [55] and Hiew [32] examined the use of recommenders to assist triagers with one of the repository-oriented decision, determining if a new report duplicates an existing report. Runeson and colleagues used a natural language approach and reported being able to find 66% of the duplicates for reports in a corporate bug repository. Hiew's approach also used natural language processing as well as clustering, and achieved precision and recall rates of 29% and 50% respectively.

Chapter 8

Conclusions

The use of a bug repository in the software development process has a number of benefits for a project. However, part of the cost of using a bug repository is the need for the reports to be organized through the triage process. The organizational decisions made by triagers can be divided into two types. The first type includes decisions that determine if a report is meaningful, such as if the report is a duplicate or is not reproducible. We call such decisions repository-oriented decisions. The second type includes decisions that organize the report for the project's development process, such as determining the product component the report affects or the developer to assign the report. We call these development-oriented decisions. Making these decisions uses resources that might better be spent improving the product rather than managing the development process.

This dissertation focuses on reducing human involvement in development-oriented decisions through a process we call the the triage-assisting recommender creation process. The work described in this dissertation makes four contributions to this field of software engineering.

First, we present a machine learning-based triage-assisting recommender creation process/framework, which we refer to as ML_{Triage}, for creating recommenders that provide suggestions to help a triager make development-oriented decisions.

Second, we show, through analytical and empirical evaluations, that recommenders with good accuracy can be created using ML_{Triage}. The analytical evaluation was conducted across five different open source projects and showed that recommenders with good accuracy could be created using ML_{Triage}. The empirical evaluation was conducted using a field study with four triagers from the UI component of the Eclipse Platform project and showed that the recommenders worked well in practice.

Third, we introduce an approach to assist in the configuration of development-oriented recommenders. This allows a project member to apply their project-specific knowledge to the configuration of R_{DO} recommenders for their project.

Lastly, we show it is possible to provide recommendations based on learning from repository information on a client rather than on a server. This allows R_{DO} recommenders

Table 8.1: Triager wish lists for triage-assisting recommenders.

Triager A	Is this a duplicate?
	When did this regress?
Triager B	How severe is this problem?
Triager C	How severe is this problem?
	Is this a duplicate?
	What operating system is this for?
	Keywords for the report.
Triager D	How severe is this problem?
	What is the business priority of this report?
	Code solutions for similar types of problems.

to be created using a smaller amount of data.

8.1 Future Work

This work has looked at one point in the software development process, bug report triage, and presented an approach to assist with one type of triage decision, development-oriented decisions. However, more work needs to be done towards assisting with triage, specifically with assisting repository-oriented decisions. There has already been some work towards assisting with one type of repository-oriented decision, detecting duplicate reports [32, 55]. Observations of developer information needs have also shown that developers spend a lot of time trying to reproduce behaviour specified in a report [39]. To the best of our knowledge, there has been no work in automating reproduction of defect behaviour.

One of the questions posed to the four Eclipse triagers during our interview was for what other fields or pieces of information would they like to have recommendations. The triagers were told that they were allowed to ask for things that did not necessarily seem feasible, but to simply provide a wish list. Table 8.1 shows their wish lists. The lists are not necessarily in order of importance to the triagers. From the table we see that three of the four triagers listed an estimation of severity of the reported problem. This is another type of R_{DO} whose creation using ML_{Triage} could be explored.

From our experiences with this work, we saw a need to provide triagers with the ability to train and use R_{DO} on their machine (i.e., a client environment) so as not to require installing software on the project's bug repository server. To this end, we created the assisted configuration approach for R_{DO} recommenders (see Chapter 5). Although we demonstrated and evaluated the approach for creating developer assignment recommenders (R_A), there is more work to be done. Firstly, more investigation is needed towards using the assisted configuration approach for other R_{DO} recommenders. Although we have presented how to use the assisted configuration approach to aid in the creation of other recommenders such as a project-component recommender (R_C) and sub-component recommender (R_S), further evaluation is necessary to determine if the assisted configuration process produces recommenders that are comparable to those created in Chapter 5. Secondly, although we provided mock-ups of the interface for assisting with R_A configuration, there is still work to be done towards testing a user interface for the assisted approach.

The results from our assisted configuration evaluation showed that repository mining techniques may be viable on less than a full repository of data. This opens exciting new directions and questions for mining software repository work, such as the applicability of mining approaches to smaller projects and helping users to understand process data extracted from repository data.

A final area of future work is with respect to the adoption of R_{DO} recommenders in to a project's triage process. One of the challenges we faced in conducting the field study was in getting triagers to try Sibyl, and in this we found we faced a paradox. Although we configured recommenders for several projects, triagers were hesitant to use the recommenders until they were proven effective. However, to demonstrate that they were effective, we needed to have triagers use them. This dissertation presents initial results that indicate the effectiveness of R_{DO} recommenders. However more work needs to be done towards the goal of adoption, specifically in two directions.

The first direction is more evaluation of the recommenders to determine whether or not the use of R_{DO} recommenders reduces the time a triager takes to triage a report. Recall that this was one of our research questions (see Chapter 4). As mentioned in Section 4.2.3, we were not able to collect enough information to determine if there was an effect. Observational studies of triagers using the recommenders along with a more precisely instrumented interface is likely needed to answer this research question. Demonstrating that the use of R_{DO} recommenders can reduce triage time will likely make the use of the recommenders more attractive to triagers.

The second direction of work towards the adoption of R_{DO} recommenders is further exploration of the range of R_{DO} recommenders that can be created using ML_{Triage}. Section 6.3, explained how ML_{Triage} might be used to create recommenders for both effort estimation and impact analysis, however the creation and evaluation of these recommenders has been left for future work. Providing more R_{DO} recommenders may also provide incentive to triagers. For example, the Eclipse Platform UI triagers became interested in a sub-component recommender after using Sibyl.

8.2 Contributions

This work makes the following contributions to the field of software engineering.

First, we presented an approach to creating R_{DO} recommenders that generalizes across types of R_{DO}. In this dissertation we investigated four types of R_{DO} recommenders:

a recommender for who to assign a report to (R_A), recommenders for which component (R_C) and sub-component (R_S) to file a report against, and a recommender for which other project members may want to be informed about progress on this report (R_I).

Second, we showed that ML$_{Triage}$ generalizes across software projects. We created R_A, R_C, and R_I recommenders for five different open source projects.

Third, we showed the recommenders created using ML$_{Triage}$ are generally accurate. This was shown in two ways. The first was an analytical evaluation of the R_A, R_C, and R_I recommenders. We further evaluated the R_A, R_C, R_S, and R_I recommenders in a field study conducted with four Eclipse platform triagers for the UI component. From this field study we confirmed our analytical results.

Fourth, this dissertation provided an exploration of using supervised and unsupervised machine learning algorithms to create R_A recommenders. We evaluated the use of conjunctive rules, Naïve Bayes, Support Vector Machines, nearest-neighbour, C4.5, and Expectation Maximization to create such a recommender. We found that Support Vector Machines creates the most accurate R_A in general.

Fifth, we presented an approach to assist project members in configuring project-specific parameters when creating a R_{DO} recommender. We demonstrated this approach by creating R_A recommenders for five projects. We showed how the approach assists a project member in determining which reports to use, how many reports to use, how to label the reports, and which labels are valid.

Finally, this dissertation provides a study of how triage is accomplished and the challenges faced by triagers for the project team of a successful open-source project. This study included information from questionnaires, triager interviews, and a field study showing how R_{DO} recommenders work in practice.

Bibliography

[1] Rakash Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, 1995.

[2] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, January 1991.

[3] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with open bug repositories. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[4] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 318–370, 2006.

[5] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 2. IEEE Computer Society, 2007.

[6] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press, 1999.

[7] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 137–143, May 2006.

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

[9] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, 2004.

[10] Bugsquad/triageguide. Web page. http://live.gnome.org/Bugsquad/TriageGuide.

[11] Christopher S. Campbell, Paul P. Maglio, Alex Cozzi, and Byron Dom. Expertise identification using email communications. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 528–531. ACM Press, 2003.

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

[13] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proc. of International Software Metrics Symposium*, page 29. IEEE Computer Society Press, 2005.

[14] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *Proc. of the 21st ACM Symposium on Applied Computing*, pages 1767- 1772. ACM Press, 2006.

[15] Peter H. Carstensen and Carsten Sorensen. Let's talk about bugs! *Scandinavian Journal of Information Systems*, 7(1):33–54, 1995.

[16] Kevin Crowston. A coordination theory approach to organizational process design. *Organization Science*, 8(2):157–175, 1997.

[17] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006.

[18] Kevin Crowston and Barbara Scozzi. Coordination practices within floss development teams: The bug fixing process. In *Computer Supported Acitivity Coordination*, pages 21–30. INSTICC Press, 2004.

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

[20] Cleidson R. B. de Souza, David Redmiles, Gloria Mark, John Penix, and Maarten Sierhuis. Management of interdependencies in collaborative software development. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 294–303, 2003.

[21] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[22] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[23] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, and Sara Gradara. An approach to classify software maintenance requests. In *Proceedings of the International Conference on Software Maintenance*, pages 93–102, 2002.

[24] Thomas G. Dietterich. Machine learning. *Encyclopedia of Cognitive Science*, 2003.

[25] Trung T. Dinh-Trong and James M. Bieman. The freebsd project: A replication case study of open source development. *IEEE Transactions on Software Engineering*, 31(6):481–494, 2005.

[26] Pedro Domingos and Michael J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *Proc. of the International Conference on Machine Learning*, pages 105–112, 1996.

[27] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 23. IEEE Computer Society, 2003.

[28] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32. IEEE Computer Society, 2003.

[29] George Forman and Ira Cohn. Learning from little: Comparision of classifiers given little training. In *Proceedings of 8th European Conference on Principles and Practices of Knowledge Discovery in Databases*, pages pp. 161–172, 2004.

[30] Les Gasser and Gabriel Ripoche. Distributed collective practices and F/OSS problem management: Perspective and methods. In *2003 Conference on Cooperation, Innovation and Technologie*, 2003.

[31] Steve 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.

[32] Lyndon Hiew. Assisted detection of duplicate bug reports. Master's thesis, University of British Columbia, 2006.

[33] Thorsten 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.

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

[35] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM Press, 2005.

[36] James Howison Kevin Crowston. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[37] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007.

[38] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[39] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proc. of the 29th Int'l Conference on Software Engineering*, pages 344–353, 2007.

[40] Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27 – 42, 2002.

[41] David Mattox, Mark T. Maybury, and Daryl Morey. Enterprise expert and knowledge discovery. In *Proceedings of the International Conference on Human-Computer Interaction*, pages 303–307. Lawrence Erlbaum Associates, Inc., 1999.

[42] David W. McDonald. Evaluating expertise recommendations. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 214–223. ACM Press, 2001.

[43] David W. McDonald and Mark S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *Proceedings of ACM Conference on Computer Supported Collaborative Work*, pages 231–240, 2000.

[44] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *Proc. of 4th Int'l Workshop on Mining Software Repositories*, page 5. IEEE Computer Society, May 2007.

[45] Tom M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997.

[46] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

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

[48] Sandro Morasca and Gunther Ruhe. A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. *Journal of Systems and Software*, 53(3):225–237, Year.

[49] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, 2003.

[50] Ross Quinlan. *C4.5: Programs for Machine Learning.* 1993.

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

[52] Christian Robottom Reis and Renata Pontin 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.

[53] Jason D. M. Rennie, Lawrence Shih, Jaime Teevan, and David R. Karger. Tackling the poor assumptions of Naïve Bayes classifiers. In *Proceedings of International Conference on Machine Learning*, pages 616–623, 2003.

[54] Gregorio Robles, Stefan Koch, and Jess M. Gonzlez-Barahona. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, May 2004.

[55] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. of the 29th Int'l Conference on Software Engineering*, pages 499–510, 2007.

[56] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):pp. 15–45, October 1997.

[57] Robert J. Sandusky and Les Gasser. Negotiation and the coordination of information and activity in distributed software problem management. In *Proceedings of the International Conference on Supporting Group Work*, pages 187–196. ACM Press, 2005.

[58] Robert J. Sandusky, Les Gasser, and Gabriel Ripoche. Bug report networks: Varieties, strategies, and impacts in a F/OSS development community. *Proceedings of 1st Int'l Workshop on Mining Software Repositories*, pages 80–84, 2004.

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

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

[61] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the International Workshop on Mining Software Repositories*, pages 1–5. ACM Press, 2005.

[62] Xiaodan Song, Belle L. Tseng, Ching-Yung Lin, and Ming-Ting Sun. Expertisenet: Relational and evolutionary expert modeling. In *Proceedings of User Modeling*, pages 99–108. Springer Berlin, 2005.

[63] Lynn A. Streeter and Karen E. Lochbaum. An expert/expert-locating system based on automatic representationof semantic structure. In *Proceedings of the Fourth Conference on Artifical Intelligence Applications*, pages 345–350, 1988.

[64] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[65] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of 4th Int'l Workshop on Mining Software Repositories*, 2007.

[66] Chadd C. Williams and Jeffrey K. Hollingsworth. Bug driven bug finders. In *Proceedings of the 2004 International Workshop on Mining Software Repositories*, pages 70–74, May 2004.

[67] Ian H. Witten and Timothy C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Transactions on Information Theory*, 37(4):1085–1094, July 1991.

[68] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools with Java Implementations.* 2000.

[69] Dawit Yimam-Seid and Alfred Kobsa. Expert-finding systems for organizations: Problem and domain analysis and the DEMOIR approach. *Journal of Organizational Computing and Electronic Commerce*, 13(1):1–24, 2003.

[70] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[71] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages pp.563–572, 2004.

Appendix A

Labeling Heuristics

This appendix presents the heuristics that were used for labeling the reports for the analytical evaluation of developer recommenders from Section 4.1.2.

A.1 Eclipse Platform Project

These heuristics are extensions of the ones given by Čubranić and Murphy [19].

1. If the report has not been resolved (i.e., NEW, UNCONFIRMED, REOPENED) label the report with the name of the developer who was last assigned to it.

2. If the report is marked as ASSIGNED, then a developer has taken responsibility for fixing the report and the report is labeled with their name.

3. If the report was resolved by the assigned-to person, label the report with the assigned-to person.

4. If the report was not resolved by the person who filed the report or who was assigned to the report, label the report with the name of the developer who resolved it.

5. If the report is marked as FIXED, regardless of who resolved it, label the report with the person who marked it as such.

6. If the report was resolved as not FIXED (i.e., WONTFIX, WORKSFORME, INVALID) by the person who filed the report and was not assigned to it, label the report with the name of the first person who responded to the report. This heuristic covers cases where the submitter retracts a report such as after being asked for more information and not being able to reproduce the problem.

7. If the report was resolved as not FIXED by the person who filed the report and was not assigned to it, and no one responded to the bug, then the report cannot be labeled. This heuristic covers cases when a bug is submitted by error, such as the reporter

not knowing the proper operation of Eclipse, and retracts the bugs before anyone else responds to it.

8. If the report was resolved as LATER or WONTFIX by the person who filed the report and was not assigned to it, label the report with the name of the reporter. This heuristic covers reports that are to-do items for the developer.

9. If the report is marked as a DUPLICATE, label the report with the label of the report of which this report is a duplicate of.

A.2 Mozilla Projects

As all of the Mozilla projects use the same development process, we used the same heuristics for both the Firefox and Bugzilla projects.

1. If the report is marked as NEW, UNCONFIRMED, or REOPENED then no one has taken responsibility for the report and it is unknown to which developer it will be assigned. The report is labeled as unclassifiable.

2. If a report is resolved as WORKSFORME, it was so marked by the person doing the bug triage, and it is unknown which developer would have been assign the report. The report is labeled as unclassifiable.

3. If the report is marked as ASSIGNED, then a developer has taken responsibility for fixing the report and the report is labeled with their name.

4. If the report is resolved as FIXED and the report has attachments which were approved by a reviewer, then

   (a) If there is only one submitter of approved patches, label the report with their name.

   (b) If there is more than one submitter of approved patches, choose the name of the developer who submitted the most patches.

   (c) If the submitter(s) of the approved patches cannot be determined, then label the report with the person who is assigned to the report.

5. If the report was marked as FIXED without an attached patch, then it was likely a regression and the report is labeled with the name of the person who marked the report as fixed.

6. If the report is marked as INVALID and the person who resolved the report was not the person who submitted the report, label the report with the name of the developer who made the decision.

7. If the report was marked as INVALID by the person who submitted the report, the report was likely submitted in error and it is unknown which developer would have been assign the report. The report is labeled as unclassifiable.

8. Reports that are marked as WONTFIX are often resolved after some discussion and the developers reach a consensus. It is unknown which developer would have fixed the bug, so the report is labeled as unclassifiable.

9. If the report is marked as a DUPLICATE, label the report with the label of the report of which this report is a duplicate of.

A.3 gcc Project

1. If the report is NEW, UNCONFIRMED, or REOPENED then no developer is responsible for it and it is unknown who will be assigned. The report is labeled as unclassifiable.

2. If the report is marked as ASSIGNED, then a developer has taken responsibility for fixing the report and the report is labeled with their name.

3. If the report is marked as WAITING or SUSPENDED, then a developer is waiting for more information or some other event to occur, such as another bug being fixed, before this bug can be resolved. As a developer is in the process of resolving the problem, label the report with their name.

4. If a report is resolved as FIXED, then label the report with the name of the person who marked the report as fixed.

5. If the report was resolved as INVALID, WONTFIX, or WORKSFORME by the submitter then the report was likely submitted in error and it is unknown who would have been assigned the report. The report is labeled as unclassifiable.

6. If the report was resolved as INVALID, WONTFIX, or WORKSFORME by someone other than the submitter, then label the report with the developer name that made that determination.

7. If the report was marked as INVALID and report was for the spam component then the report was the result of actions by a spammer and is labeled as unclassifiable.

8. While the report status CLOSED is not valid for the gcc project, some bug activity logs show that that a bug was marked as such. These appear to be old reports that are in the repository so that other reports can reference them, such as duplicates. Such reports are labeled with the name of the developer who last changed the status of the report.

9. If the report is marked as a DUPLICATE, label the report with the label of the report of which this report is a duplicate of.

A.4 Mylyn Project

1. If the report is NEW or REOPENED then no developer is responsible for it and it is unknown who will be assigned. The report is labeled as unclassifiable.

2. If the report is marked as ASSIGNED, then a developer has taken responsibility for fixing the report and the report is labeled with their name.

3. If the report is marked as a DUPLICATE, label the report with the label of the report of which this report is a duplicate of.

4. If the report is marked as FIXED, regardless of who resolved it, label the report with the person who marked it as such.

5. If the report was resolved as not FIXED (i.e., WONTFIX, WORKSFORME, INVALID) by the person who filed the report and was not assigned to it, label the report with the name of the first person who responded to the report. This heuristic covers cases where the submitter retracts a report such as after being asked for more information and not being able to reproduce the problem.

6. If the report was resolved as not FIXED by the person who filed the report and was not assigned to it, and no one responded to the bug, then the report cannot be labeled. This heuristic covers cases when a bug is submitted by error, such as the reporter not knowing the proper operation of Eclipse, and retracts the bugs before anyone else responds to it.

7. Reports that are marked as LATER are often resolved after some discussion and the developers reach a consensus. It is unknown which developer would have fixed the bug, so the report is labeled as unclassifiable.

Appendix B

Labeling Heuristics from Assisted Configuration

This appendix presents the various configurations used during the evaluation of the assisted configuration approach (see Chapter 5).

Table B.1: Heuristics used for the Eclipse project.

NF(V)?	Who last marked FIXED
(N)?FV	Who last marked FIXED
NFV	Who last marked FIXED
NF(C)?	Who last marked FIXED
NF	Who last marked FIXED
ND	Label of report that is duplicated
NAF(V)?	Who last marked FIXED
(N)?AF(V)?	Who last marked FIXED
(N)?ANF(V)?	Who last marked FIXED
NANF(V)?	Who last marked FIXED

Table B.2: Heuristics used for the Firefox project.

UD	Label of report that is duplicated
(U)?NA(F)?	Who attached last patch, Value of assigned-to field
(NA)+(F)?	Who attached last patch, Value of assigned-to field
NA(F)?	Who attached last patch, Value of assigned-to field
(UI)+(V)?	(No Data Sources Selected)
(U)?NA	Value of assigned-to field
UI(V)?	Value of assigned-to field
NA	(No Data Sources Selected)
NF(V)?	Who attached last patch, Who last marked FIXED
UI	(No Data Sources Selected)

Table B.3: Heuristics used for the gcc project.

U(NA)+(F)?	Who attached last patch, Value of assigned-to field
(U)?(NA)+(F)?	Who attached last patch, Value of assigned-to field
(U)?NA(F)?	Who attached last patch, Value of assigned-to field
UNA(F)?	Who attached last patch, Value of assigned-to field
U(NA)+F	Who last marked FIXED, Value of assigned-to field
(U)?(NA)+F	Who last marked FIXED, Value of assigned-to field
UNAF	Who last marked FIXED, Value of assigned-to field
(U)?NAF	Who last marked FIXED, Value of assigned-to field
UD(V)?	Label of report that is duplicated
UA(F)?	Who last marked FIXED, Value of assigned-to field

Table B.4: Heuristics used for the Mylyn project.

NF	Who last marked FIXED
ND(C)?	Label of report that is duplicated
ND	Label of report that is duplicated
(N)?(FO)+F	Who last marked FIXED
N(FO)+(F)?	Who last marked FIXED, Who last marked RESOLVED
(N)?(FO)+(F)?	Who last marked FIXED, Who last changed the status
N(FO)+F	Who last marked FIXED
NFO(F)?	Who last marked FIXED, Who last changed the status
NFOF	Who last marked FIXED
(N)?FOF	Who last marked FIXED

Table B.5: Heuristics used for the Bugzilla project.

(U)?NA(F)?	Who attached last patch, Who last marked FIXED, Value of assigned-to field
(NA)+(F)?	Who attached last patch, Who last marked FIXED, Value of assigned-to field
(U)?NAF	Who attached last patch, Who last marked FIXED, Value of assigned-to field
NA(F)?	Who attached last patch, Who last marked FIXED, Value of assigned-to field
(NA)+F	Who attached last patch, Who last marked FIXED, Value of assigned-to field
NAF	Who attached last patch, Who last marked FIXED, Value of assigned-to field
(N)?AF	Who attached last patch, Who last marked FIXED, Value of assigned-to field
UNA(F)?	Who attached last patch, Who last marked FIXED, Value of assigned-to field
UD(V)?	Label of report that is duplicated
UNAF	Who attached last patch, Who last marked FIXED, Value of assigned-to field

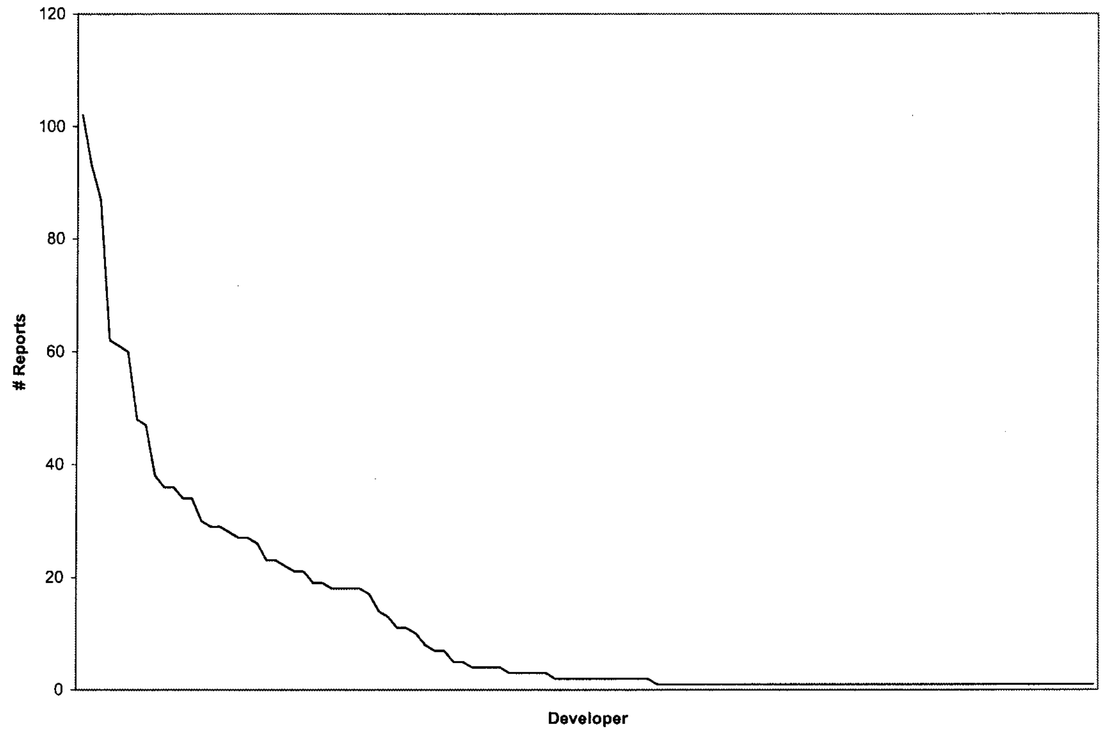


Figure B.1: Developer resolution graph for Eclipse.

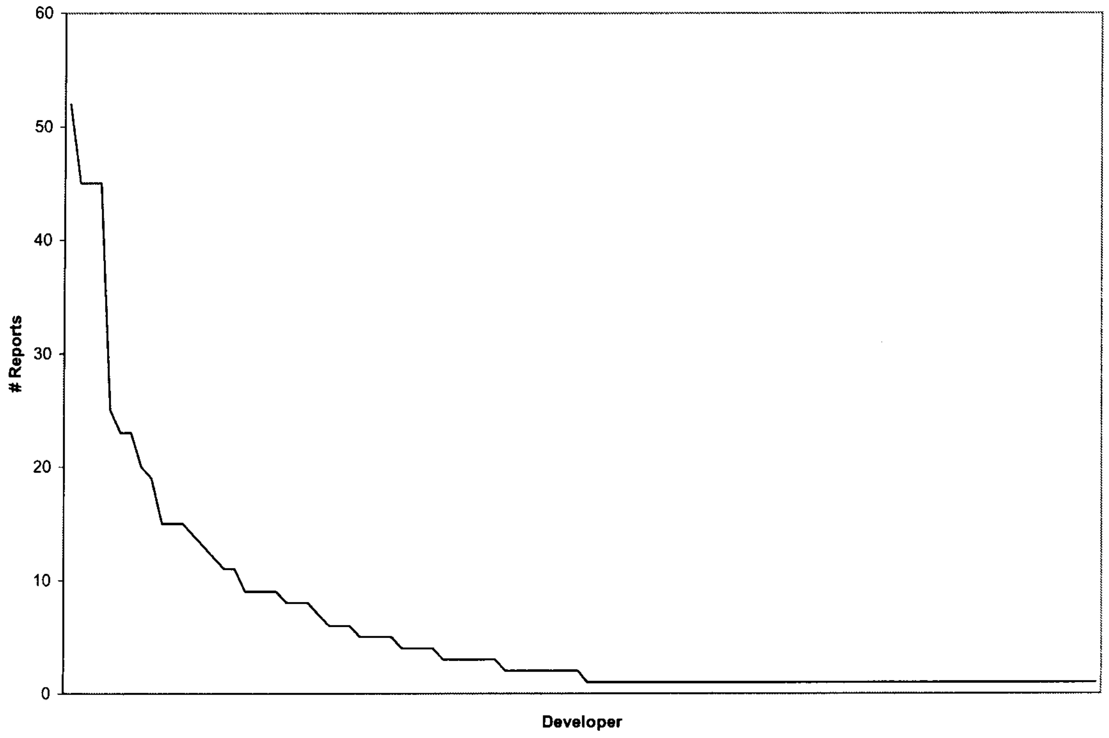


Figure B.2: Developer resolution graph for Firefox.

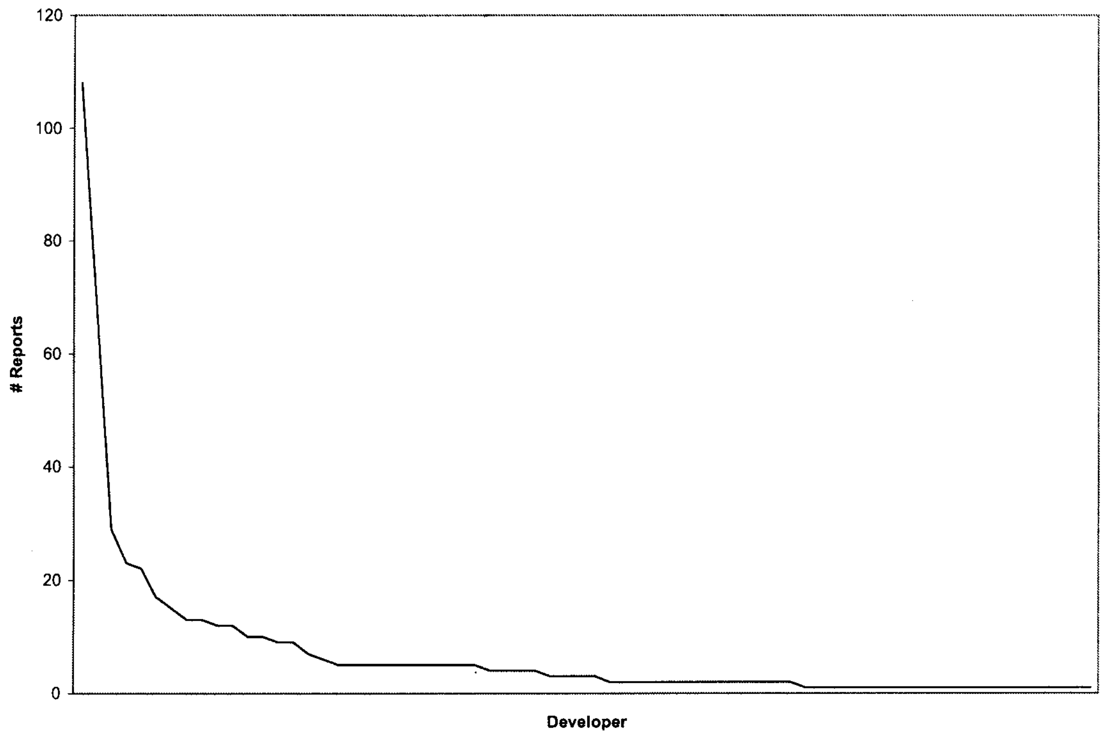


Figure B.3: Developer resolution graph for gcc.

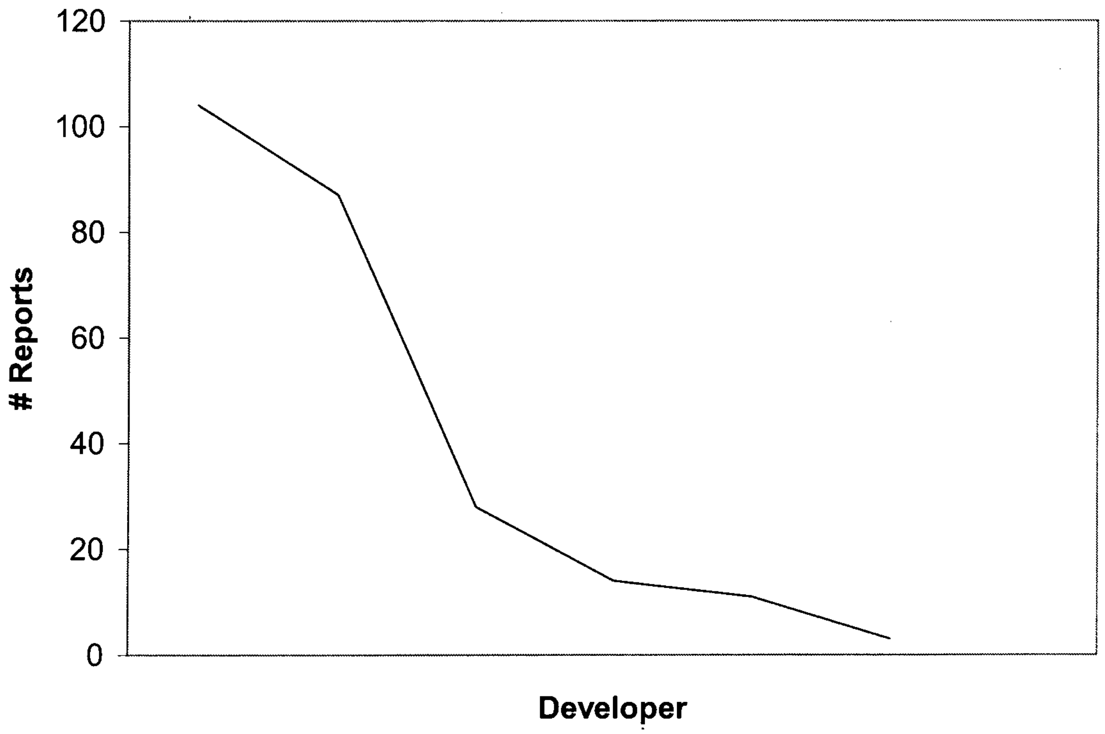


Figure B.4: Developer resolution graph for Mylyn

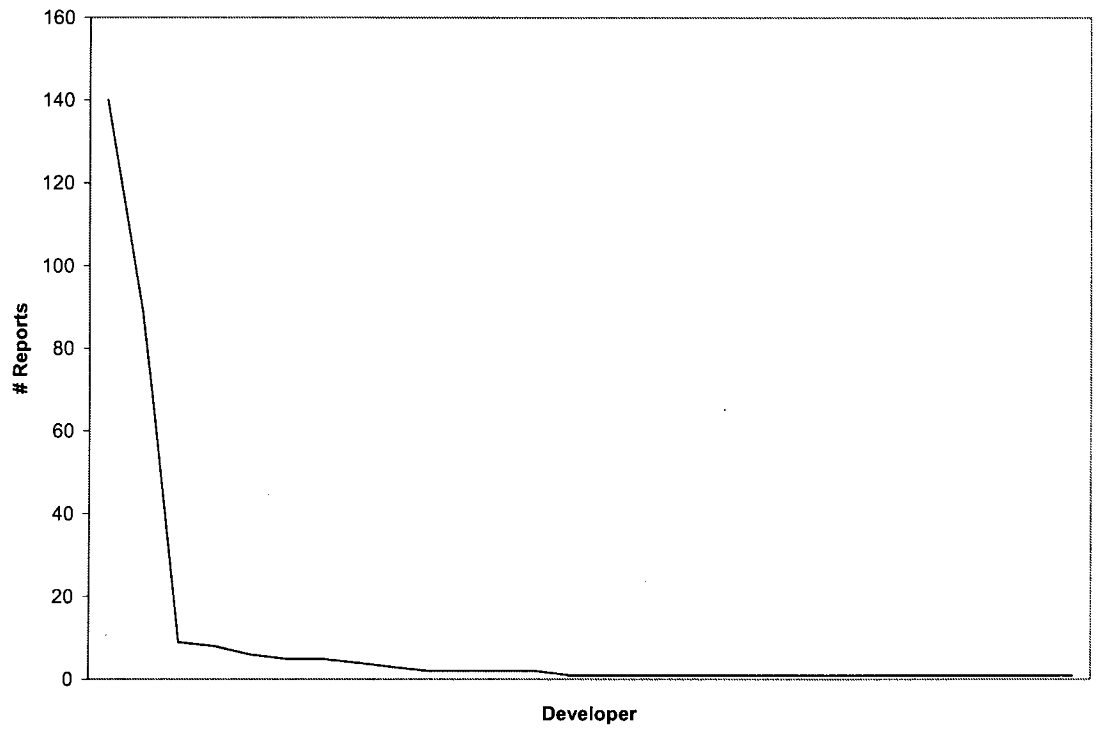
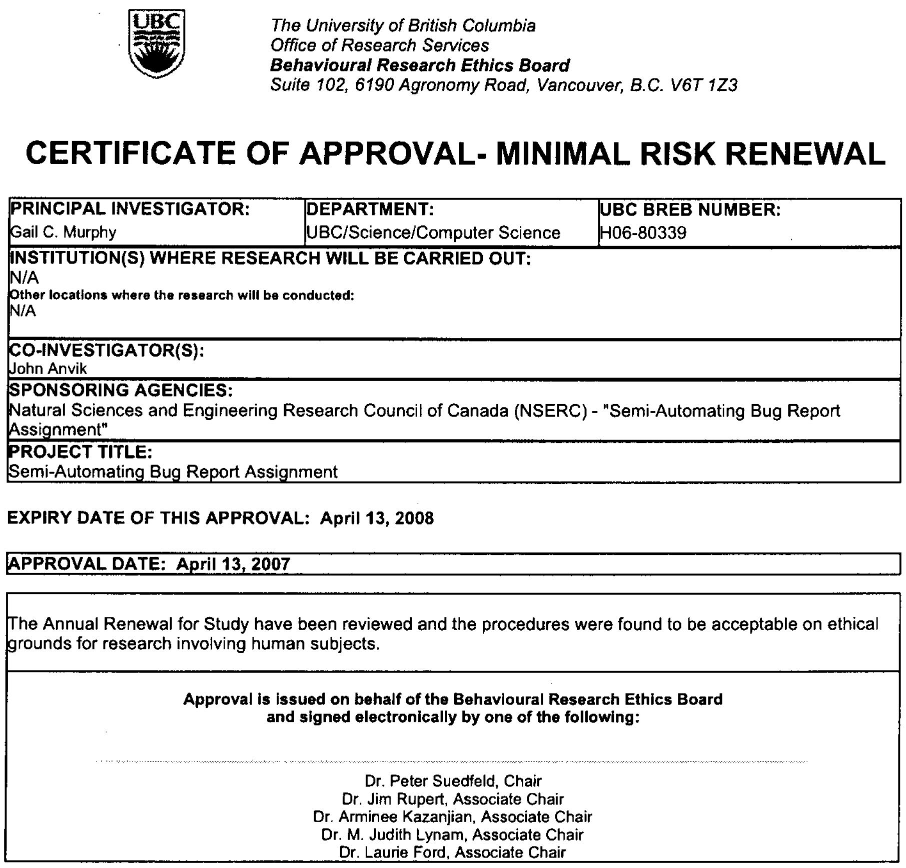


Figure B.5: Developer resolution graph for Bugzilla

Appendix C

Ethics Certificate



The University of British Columbia
Office of Research Services
Behavioural Research Ethics Board
Suite 102, 6190 Agronomy Road, Vancouver, B.C. V6T 1Z3

CERTIFICATE OF APPROVAL- MINIMAL RISK RENEWAL

PRINCIPAL INVESTIGATOR:	DEPARTMENT:	UBC BREB NUMBER:
Gail C. Murphy	UBC/Science/Computer Science	H06-80339

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:
N/A
Other locations where the research will be conducted:
N/A

CO-INVESTIGATOR(S):
John Anvik

SPONSORING AGENCIES:
Natural Sciences and Engineering Research Council of Canada (NSERC) - "Semi-Automating Bug Report Assignment"

PROJECT TITLE:
Semi-Automating Bug Report Assignment

EXPIRY DATE OF THIS APPROVAL: April 13, 2008

APPROVAL DATE: April 13, 2007

The Annual Renewal for Study have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board and signed electronically by one of the following:

Figure C.1: Ethics Certificate