Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions

JOHN ANVIK, Central Washington University
GAIL C. MURPHY, University of British Columbia

A key collaborative hub for many software development projects is the bug report repository. Although its use can improve the software development process in a number of ways, reports added to the repository need to be triaged. A triager determines if a report is meaningful. Meaningful reports are then organized for integration into the project's development process.

To assist triagers with their work, this article presents a machine learning approach to create recommenders that assist with a variety of decisions aimed at streamlining the development process. The recommenders created with this approach are accurate; for instance, recommenders for which developer to assign a report that we have created using this approach have a precision between 70% and 98% over five open source projects. As the configuration of a recommender for a particular project can require substantial effort and be time consuming, we also present an approach to assist the configuration of such recommenders that significantly lowers the cost of putting a recommender in place for a project. We show that recommenders for which developer should fix a bug can be quickly configured with this approach and that the configured recommenders are within 15% precision of hand-tuned developer recommenders.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*Productivity*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces; I.2.1 [**Artifical Intelligence**]: Applications and Expert Systems; I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms: Management, Design

Additional Key Words and Phrases: Bug report triage, recommendation, task assignment, machine learning, configuration assistance

ACM Reference Format:
Anvik, J. and Murphy, G. C. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. 20, 3, Article 10 (August 2011), 35 pages.
DOI = 10.1145/2000791.2000794 http://doi.acm.org/10.1145/2000791.2000794

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. This database is often called an *issue tracking system* or *bug repository*. The use of a bug repository can improve the development process in a number of ways including tracking the work that needs to be done, allowing developers who are geographically distributed

This article is a revised and extended version of a paper presented at the 2006 International Conference on Software Engineering.
This research was supported by the National Sciences and Engineering Research Council of Canada (NSERC) and IBM.
Authors' addresses: J. Anvik, Department of Computer Science, 400 E. University Way, Ellensburg, WA 98926-7520; email: janvik@cwu.edu; G. C. Murphy; Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, BC, Canada, V6T 1Z4; email: Murphy@cs.ubc.ca.

DOI 10.1145/2000791.2000794 http://doi.acm.org/10.1145/2000791.2000794

to communicate about project development [Mockus et al. 2002; Sandusky and Gasser 2005], tracking the evolution of the project by the number of outstanding bug reports [Carstensen and Sorensen 1995; de Souza et al. 2003], and improving the quality of software produced [Raymond 1999; Mockus et al. 2002; Crowston et al. 2006].

However, the use of a bug repository is not without cost. Each report submitted to a bug repository must be *triaged*[1] by a project member to determine if it describes a valid problem [Reis and de Mattos Fortes 2002] and if so, how the report should be categorized for handling through the development process [Carstensen and Sorensen 1995; Di Lucca et al. 2002]. These triage activities divert effort away from improving the product toward managing the project.

The *triager*, the project member who triages the report, has two primary goals. The first goal is to have the repository contain the smallest set of best reports for the project so that development effort can focus on the reports that lead to product improvements. For example, a triager may decide that the problem described by a new report has already been identified by an existing report; the new report can be marked to indicate that no further work is needed for it. We call triage decisions made with the goal of creating the smallest best set of reports *repository-oriented decisions*.

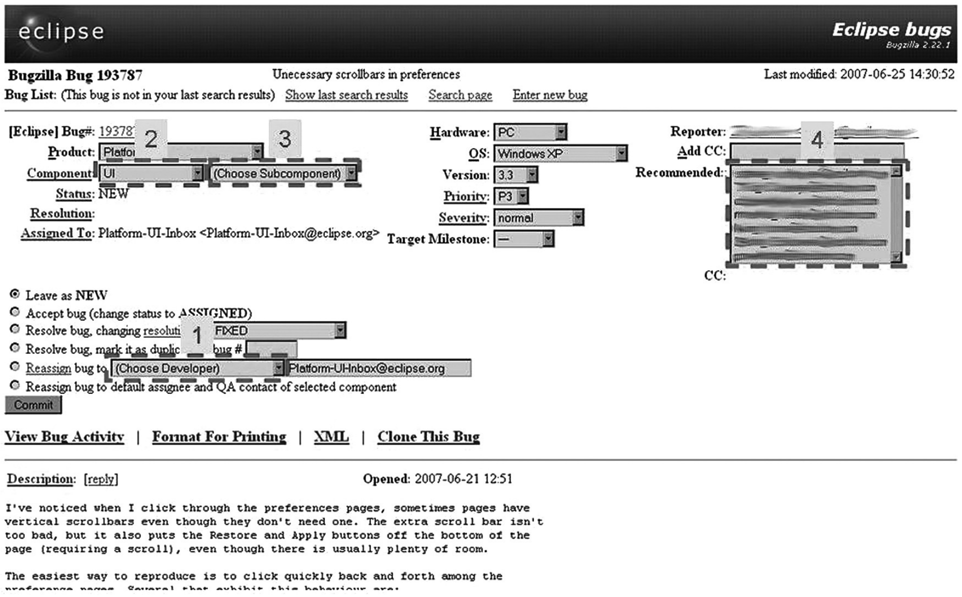
A second goal of the triager is to organize the reports for integration into the development process. Reports may be organized in a variety of ways, such as by the product component[2] it affects or the developer assigned to the report. We refer to decisions made with this goal in mind as *development-oriented decisions*. Typically, development-oriented decisions are made after repository-oriented decisions. However, the two decisions may be intermixed.

Every decision made during triage requires a project member to expend time and effort to make a decision. Sometimes the decision can be made based on knowledge immediately available; in these cases, the decision cost is low. Other times the decision cost is high as the triager must gather information through various means, including searching the repository and sifting through many results to learn if the problem has already been described, investigating how similar problems have been categorized in the past, seeking out and consulting with other project members, and so forth. Many of these information gathering approaches require significant human involvement, increasing the likelihood of mistakes. When mistakes are made, more time and effort of the triager or developers is required to overcome the mistakes.

The cumulative cost of making these decisions can be substantial. For example, the Eclipse Platform project[3] received an average of thirty new reports a day over the period of April to June 2007. Based on observations, we have made of open source developers, it appears to take roughly five minutes on average for a triager to perform the initial triage on a report. As a result, on average, approximately two and a half person-hours per day was spent for the Eclipse Platform project triaging reports during the aforementioned time period. We have observed that approximately 25% of Eclipse reports are later reassigned, which results in additional triage effort occurring later in the process.

To reduce the time and effort expended by triagers, we describe an approach in which a triager is presented with a list of suggestions from *recommenders* for various

[1]triage: A process in which things are ranked in terms of importance or priority. (Merriam-Webster Dictionary).
[2]We use the term, *component*, to refer to a specific piece of product functionality, such as the user interface, network protocols, or business logic.
[3]Eclipse is an open-source platform for integrating software components (www.eclipse.org, verified 3/21/09). The Eclipse Platform project within the Eclipse ecosystem provides the base infrastructure for component integration.



Bugzilla Bug 193787 Unecessary scrollbars in preferences Last modified: 2007-06-25 14:30:52
Bug List: (This bug is not in your last search results) Show last search results Search page Enter new bug

[Eclipse] Bug#:	193787	Hardware:	PC	Reporter:	[redacted]
Product:	Platfor	OS:	Windows XP	Add CC:	
Component:	UI (Choose Subcomponent)	Version:	3.3	Recommended:	[redacted]
Status:	NEW	Priority:	P3		
Resolution:		Severity:	normal		
Assigned To:	Platform-UI-Inbox <Platform-UI-Inbox@eclipse.org>	Target Milestone:	—	CC:	

- [✓] Leave as NEW
- Accept bug (change status to ASSIGNED)
- Resolve bug, changing resoluti FIXED
- Resolve bug, mark it as duplic bug #
- Reassign bug to (Choose Developer) Platform-UI-Inbox@eclipse.org
- Reassign bug to default assignee and QA contact of selected component

Commit

View Bug Activity | Format For Printing | XML | Clone This Bug

Description: [reply] Opened: 2007-06-21 12:51

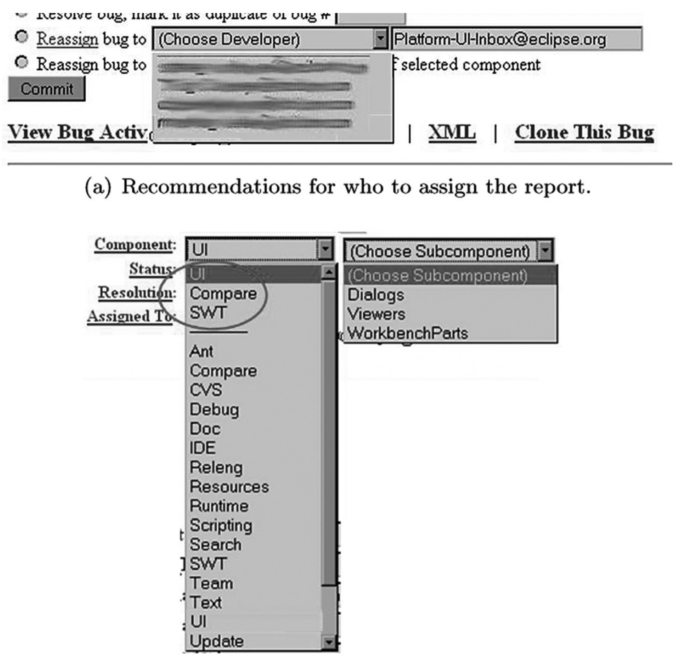
```
I've noticed when I click through the preferences pages, sometimes pages have
vertical scrollbars even though they don't need one. The extra scroll bar isn't
too bad, but it also puts the Restore and Apply buttons off the bottom of the
page (requiring a scroll), even though there is usually plenty of room.

The easiest way to reproduce is to click quickly back and forth among the
```

Fig. 1. Example of a bug report with recommendations.

triage decisions. In this way, human involvement in triage is transformed by moving the triager's role from gathering information prior to making a decision to confirming a suggestion from the recommender. To make this idea concrete, Figure 1 shows the web interface to the Eclipse bug repository that has been augmented, based on the work reported in this article, with three recommenders: a *developer recommender* that suggests to whom to assign the report (Figure 1-1), a *component (and subcomponent) recommender* that suggests the appropriate values for these fields in the report Figures 1-2 and 1-3, respectively), and an *interest recommender* that suggests which other project members may want to be aware of the report (Figure 1-4).

In making the decision whom to assign to the report, the Eclipse triager selects from the developer recommender drop-down box (Figure 2(a), note the names have been obscured for privacy reasons). For this report, the triager is presented with four names from a possible list of forty-four Eclipse developers. Similarly, when the triager uses the project component drop-down box (Figure 2(b)-left) three recommendations are displayed from the possible eighteen Eclipse Platform components. In this case, the triager selects the recommended UI component and is provided with a drop-down box for subcomponent recommendations (Figure 2(b)-right), which displays three recommendations from the possible fifty-six UI subcomponents. Finally, the triager is presented with seven names from a list of fifty-eight individuals that may be interested in being informed about the progress of this report (Figure 1-4). This example shows how recommendations can substantially cut down on the possible answers made during triage activities.

This article presents an approach for the creation of development-oriented recommenders. We discuss how to apply the approach to create and configure the three kinds of recommenders shown in Figure 1. The approach we present extends our previous work on a developer recommender [Anvik et al. 2006] by generalizing the approach to additional development-oriented recommenders (i.e., component and interest recommenders). The work we report on in this paper also produces recommenders with higher precision and recall. To simplify the presentation, the majority of this article



(a) Recommendations for who to assign the report.

(b) Product component and subcomponent recommendations.

Fig. 2. Recommendations for development-oriented decisions.

uses the example of a developer recommender to illustrate the approach and show it has value. A brief presentation of the component and interest recommenders that can also be created with the approach is provided later in this article (Section 6.2).

We also describe in this article how the introduction of recommenders into the bug triage process for a project introduces additional project overhead. Specifically, for a project to incorporate a recommender into its development process, the recommender needs to be configured for the particular project and that configuration needs to be maintained. For example, creating a developer recommender requires creating heuristics for determining who resolved a particular bug report; the applicable heuristics depend on how a project team uses a bug repository. In our experience, determining the project-specific information to create a developer recommender for a new project typically requires between a half to a full day depending on the complexity of the project's development process. These heuristics must be checked periodically to ensure a recommender is performing well. As more recommenders are introduced into the project's development process, this overhead can become burdensome. To lower the overhead of creating and maintaining development-oriented recommenders, we present two new techniques for assisting a project member in configuring development-oriented recommenders for their specific project. We show how these techniques lower the cost of configuration to the order of minutes instead of hours and show that a recommender configured semi-automatically using the techniques performs within an acceptable range compared to a hand-tuned recommender.

We begin this article with a survey of other work about the creation of bug report triage recommenders in Section 2. We then present our approach to creating development-oriented recommenders in Section 3, using a developer recommender as an example. Next, Section 4 presents an evaluation of the approach. As the recommender creation process requires project-specific configuration, we then present in Section 5 our techniques for assisting a triager with this configuration. We discuss other aspects of the creation and configuration processes in Section 6 and discuss how our approach might apply to repository-oriented decisions and future directions for work in this area in Section 7. We summarize this article in Section 8.

2. RELATED WORK

Several earlier works have considered the creation of triage recommenders for development-oriented decisions.

Both Čubranić and Murphy [2004] and Canfora and Cerulo [2006] proposed approaches to creating developer recommenders, in which either a Naïve Bayes algorithm (Čubranić and Murphy) or a technique very similar to a Naïve Bayes algorithm (Canfora and Cerulo) is used. The two approaches differed in that Čubranić and Murphy used a text categorization approach and Canfora and Cerulo used information retrieval techniques. The text categorization approach achieved 30% precision for the Eclipse project and the information retrieval approach achieved 20% precision for the Mozilla project. In this article, we show how substantially higher precision rates can be achieved, on the order of 70% to 98%, with our approach and we show how this approach also applies to other development-oriented decisions.

Similar to our work, Di Lucca et al. [2002] investigated the use of different machine learning algorithms to recommend which of eight maintenance teams to assign a report for a single project. They too found that the Support Vector Machines and Naïve Bayes algorithms created good recommenders. In their work, they achieved good precision (over 80%) for a constrained situation: the recommender they created assigned reports to one of eight development teams for data from a commercial development project. This recommender is similar to a component recommender in our approach. With our approach, we achieve less accuracy (measured in terms of recall) for a component recommender of one recommendation (45%–66%). When we consider a component recommender that makes three recommendations, the accuracy rises to the range of the Di Lucca work (75% and greater). We show our approach works for a greater number of projects (five) with more components (11–34). In addition, the reports used in the evaluation of the Di Lucca and colleagues approach were created by a technical support team and may have used a more constrained language than the reports over which we recommend that come from a diverse population.

Using a similar approach to ours, Weiß et al. [2007] 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. Their approach used a nearest-neighbor algorithm. Using their approach, 30% of their predictions were within a ±50% range of actual effort. In this article, we show that the approach further generalizes to other development-oriented triage decisions and we tackle the problem of configuring such recommenders.

Prior to specific work on development-oriented recommenders, Mockus and Votta [2000] used an automated approach to investigate the categories of software changes for large industrial systems. This investigation included a classification of software changes based on their textual description into one of five categories: corrective, adaptive, perfective, inspection or unclassified. This approach can be seen as an early version of a development-oriented recommender as this information may alter how the report is handled for processing.

Researchers have also investigated the creation of recommenders for repository-oriented decisions. The work of Runeson et al. [2007], Hiew [2006], and Wang et al.

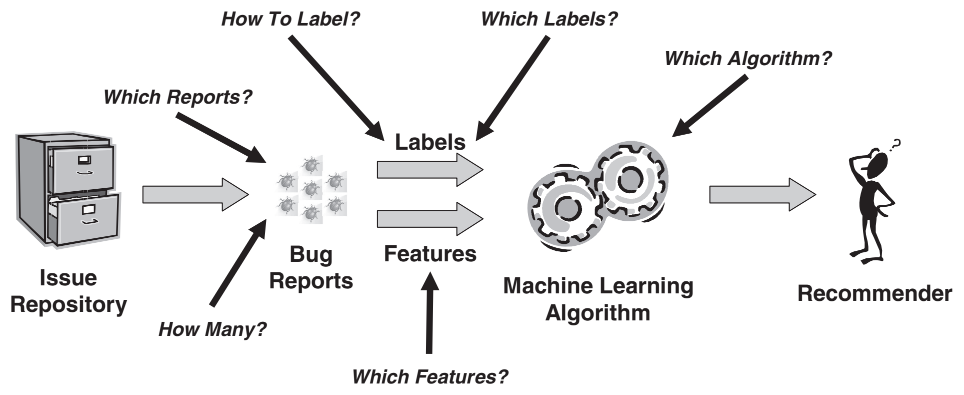


Fig. 3. Questions to be answered in creating a recommender.

[2008] have examined the use of recommenders to assist triagers with one repository-oriented decision, determining if a new report duplicates an existing report. They are similar to our work in that all three use natural language processing on the summary and description of the reports. However, all three approaches differ from our work and each other in what data is used and how the data is used. Runeson and colleagues [2007] used a natural language approach and a vector space model. They reported being able to find 66% of the duplicates for reports in a corporate bug repository. Hiew's [2006] approach also used natural language processing as well as clustering and achieved precision and recall rates of 29% and 50% respectively for the Firefox project. Wang and colleagues [2008] presented an approach that used both natural language and program execution information with a vector space approach to suggest duplicate reports. Their approach achieved recall rates of 93% for the Eclipse and Firefox projects.

Hooimeijer and Weimer [2007] presented a different approach to helping to improve the reports in a repository. They present a model based on surface features of bug reports that predicts whether or not a report will be triaged within a specified amount of time. The surface features include the reported severity, a measure of the readability of the report, the number of comments posted to the report soon after its creation, and the number of attachments to the report, amongst others. This model could be used to help order the reports for triaging, thereby serving as a form of repository-oriented recommender. In Section 7.1, we discuss the use of our approach for such repository-oriented decisions.

3. CREATING DEVELOPMENT-ORIENTED RECOMMENDERS

At a high-level, the creation of a development-oriented recommender using our approach is straightforward. Figure 3 shows an overview of the process. First, reports from a project's issue tracking system are selected automatically. 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. 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 recommendations from the set of labels.

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?

We refer to the answering of these questions as the recommender configuration process. Figure 3 depicts how these questions fit into our overall approach. Applying this process results in an instance of a recommender that can be used to make suggestions for an aspect of development-oriented triage. Our work has not considered the use of incremental techniques to update this recommender as new project data arrives. Instead, we have taken the approach of using the configuration information obtained from this process to periodically instantiate a new recommender that runs on updated data and that can be swapped with an existing instance of a recommender.

To demonstrate how these questions are answered for creating a recommender, we use the example of a developer recommender that suggests which developer should be assigned the responsibility for resolving a particular bug. A similar process for answering these questions can be used to create different recommenders (see Section 6.2).

3.1. Evaluating Recommenders

As some of the questions require experimentation to answer, we tuned our approach using data from the Eclipse Platform and Firefox[4] 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).[5]

To determine the effectiveness of our approach in creating development-oriented recommenders, we evaluate the recommenders using the standard measures of precision and recall. Precision measures how often the approach makes a relevant recommendation for a report (Eq. (1)). Recall measures how many of the recommendations that are relevant are actually recommended (Eq. (2)). For example, for a developer recommender, 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 (Eq. (3)). For the developer recommender, we use to help describe the application of our approach, we favor precision over recall, as we are looking to choose one developer with the right expertise to solve the report. At times, the recall of a developer recommender might also need to be considered to ensure an appropriate distribution of assignment of reports across developers.

$$Precision = \frac{\#\, of\ appropriate\ recommendations}{\#\, of\ recommendations\ made} \tag{1}$$

$$Recall = \frac{\#\, of\ appropriate\ recommendations}{\#\, of\ possibly\ relevant\ values} \tag{2}$$

[4]Firefox is a web browser and is available at www.mozilla.org/products/firefox (verified 3/21/09).
[5]Without loss of generality, the projects that we investigate all used the Bugzilla issue tracking system.

Table I. Date Ranges for Reports Used for Tuning a Developer Recommender

	Start Date	End Date	No. of Training Reports
Eclipse	Oct 1, 2005	May 31, 2006	6356
Firefox	Feb 1, 2006	Sept 30, 2006	3338

$$F = \frac{2 \times (precision \times recall)}{precision + recall} \tag{3}$$

3.2. Creating a Developer Recommender

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

In practice, it is not possible to answer the six questions for creating a recommender sequentially due to interdependencies. We therefore do not describe the answers to the questions sequentially, but rather present the answers in a manner that tries to minimize the interdependencies.

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. At any given moment, each report in an issue repository is at a different point in the report life-cycle supported by the issue tracking system (see Section A.2). 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 developer recommender, focusing instead on reports that have been either assigned to a developer or resolved.

As our approach for creating a recommender uses a supervised machine learning algorithm, 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 deems to not be contributing actively enough to the project 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 recommender: use a fixed quantity, select a percentage of the reports or select reports based on the time period over which they occur. We take the latter approach to provide some assurance of recency of information and use reports from the previous eight-month time period to create a developer recommender. The time frame was chosen empirically using data from the Eclipse and Firefox projects. However, this time frame depends 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 developer recommender.[6]

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 which are commonly categorized together. In the context of a developer recommender, this means determining

[6]We believe determining whether a project has a sufficient number of reports available can only be determined through experimentation. We outline a method for making such a determination elsewhere [Anvik 2007].

which reports are 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.

Each report contains a substantial amount of information in the form of pre-defined fields, free-form text, attachments, and report activity logs. Our approach uses the title (i.e., one-line summary) and full text description to characterize each report. We make this choice as these fields provide the most distinguishing information for a report; other fields such as the priority of the bug and the component to which it is assigned vary as the bug moves through the fixing process. 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 title and description, the text must be converted into a feature vector. We follow a standard text categorization approach [Sebastiani 2002] of first removing all stop words[7] and nonalphabetic tokens [Baeza-Yates and Ribeiro-Neto 1999]. These choices mean that potentially useful information, such as references to version numbers, may be excluded. These choices also mean references to source code in the text, such as the inclusion of stack traces, are treated as regular (albeit potentially odd) words. We also chose not to apply stemming[8] to the words because earlier work indicated that it had little effect [Čubranić and Murphy 2004]. We leave the determination of the influence of these choices on our approach to future work.

Words that remain after this preprocessing 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., combined title and description) length, intra-document frequency and inter-document frequency [Rennie et al. 2003]. 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. Full details about the computation of feature vectors are provided elsewhere [Anvik 2007].

3.2.4. How to Label the Reports. To train the the recommender, we need to provide a set of reports that are labeled with the name of the category to which they belong. For a developer recommender, this means determining 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 more complex because projects tend to use the status and assigned-to fields of a report for other uses. 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 the report is first assigned to a default e-mail address for a project component before being 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 II shows example heuristics for the Eclipse Platform and Firefox projects. The

[7]Stop words are functional words such as "a", "the", and "of", which do not convey meaning.
[8]Stemming identifies grammatical variations of a word, such as "see", "seeing", and "seen", and treats them as a being the same word.
[9]Both Eclipse and Firefox use the Bugzilla issue tracking system and a user name in the Bugzilla system is an e-mail address.

Table II. 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 any state but 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.

Table III. Size of Data Set, Training Set, and Testing Set Used for Process Tuning

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

full set of heuristics used for the various projects that we investigated is available on-line.[10]

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 found from the set of training reports. However, before the valid labels can be found, the set of training reports must be filtered to remove those reports that cannot be classified by the project's labeling heuristics and those reports that overlap between the training and the test set. Table III shows the effect of this filtering on the data used to tune the approach.

Once the reports that cannot be labeled by the heuristics are removed, the remaining reports provide a set of possible labels. Having learned the set of possible labels, we then calculate 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 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, whom we refer to as *active developers* in this paper.

To determine an appropriate threshold, we evaluated the effect of choosing different activity thresholds on the precision and recall of a developer recommender 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.

Table IV 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[11] over the most recent three months.

[10]See www.cs.ubc.ca/labs/spl/projects/bugTriage/assignment/heuristics.html.
[11]Eighteen reports corresponds to solving an average of six reports each month for three months.

Table IV. Precision, Recall and F-Measure when Using Developer Profile Filtering

		# Active Dev.		Precision/Recall/F as percentages	
		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	69/10/20	74/12/22

The next two columns show the number of developers considered active for the Eclipse and Firefox projects when using the given threshold. The final two columns present the precision, recall and F-measure when a developer recommender created using the given threshold makes one recommendation.

Table IV shows four things. First, the table shows that using the most recent three months removed 22 (21%) and 133 (37%) developer names from the Eclipse and Firefox recommendation pool, respectively. Second, using a threshold value of an average of one resolution per developer per month for the three month period further reduced the number of names in the recommendation pools an additional 68 (39%) and 121 (31%). Third, given that the precision and recall values do not significantly change as these names are pruned indicates that the recommendation pool maintains an appropriate set of developers. Last, this data confirms that using the most recent three months is appropriate for activity profiling.

As there is no significant change in the performance of the recommender, any value between one and eighteen 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.

3.2.6. Which Algorithm. There are a variety of machine learning algorithms that can be used to create a triage-assisting recommender. Machine learning algorithms fall under three categories: supervised learning, unsupervised learning, and reinforcement learning [Mitchell 1997]. 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.

To determine an appropriate algorithm for creating a developer recommender, we evaluated the effect of six different algorithms for the Eclipse and Firefox projects. The algorithms we investigated were Naïve Bayes [John and Langley 1995], Support Vector Machines [Gunn 1998], C4.5 [Quinlan 1993], Expectation Maximization [Dempster et al. 1977], conjunctive rules [Witten and Frank 2000], and nearest neighbour [Aha et al. 1991].[12] We chose these algorithms as they cover the different categories of supervised machine learning algorithms [Witten and Frank 2000], with Expectation Maximization providing an example of using unsupervised learning.

We created developer recommenders using the different machine learning algorithms and determined their precision, recall and F-measure.[13] The precision, recall and F-

[12]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 3/21/09).

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

Table V. Precision, Recall, and F-Measure (as Percentages) of a Developer Recommender Using Different Machine Learning Algorithms

# Recommendations	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 VI. Precision, Recall, and F-Measure (as Percentages) of a Developer Recommender when Using a Nearest-Neighbor Algorithm

# Recommendations	k = 1		k = 5		k = 10		k = 20	
	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)
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

measure of the recommenders created using five of the six algorithms is presented in Table V. For each algorithm, the table shows the measures that result for recommendation lists from size one to size three. As the seventh algorithm, nearest-neighbor, has an additional parameter—the number of neighbors to consider—Table VI shows the precision, recall and F-measure for this algorithm. For the nearest-neighbor algorithm, we explored the use of the nearest instance, and the five, ten, and twenty nearest instances. As for the other algorithms, we also show for the nearest-neighbor algorithm, the measures that result for varying sizes of recommendation lists. As we mentioned earlier, for a developer recommender, 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 developer recommenders 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 developer recommender.

3.3. Sibyl: An Implementation of the Approach

We integrated our triage-assisting recommendation approach into a web tool for the Bugzilla issue tracking system. Figure 1 from Section 1 showed this tool, called Sibyl, for a bug report from the Eclipse project. Sibyl works as a proxy to the actual web service providing the issue repository. When a report was accessed via Sibyl, Sibyl would fetch the report from the actual issue repository, compute the appropriate recommendations, augment those recommendations into the interface, and display the report. Further details about the implementation of Sibyl are available in Anvik [2007].

4. EVALUATION

Our hypothesis is that the time and effort required for development-oriented bug triage decisions can be reduced using a machine-learning-based approach. To investigate this hypothesis, we consider two research questions.

R1 Can the approach create recommenders that make accurate recommendations?
R2 Can the human triagers make use of information recommended by the approach?

Table VII. Overview of the Five Projects Used for Analytic Evaluation

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

To investigate R1, we conducted laboratory experiments to evaluate developer recommenders created using our approach for three open source projects (Section 4.1). To investigate R2, we conducted a field study, in which four human triagers from the Eclipse project used Sibyl over a four-month period (Section 4.2). Similar to our description of the approach in the previous section, we focus on the evaluation of a particular kind of development-oriented recommender—a developer recommender. Section 6.2 provides details on the evaluation of other kinds of development-oriented recommenders.

4.1. Laboratory Experiments

Our laboratory experiments consisted of creating and training a recommender for a project on several months of data and evaluating the performance of the recommender on the following month of project data using the standard measures of precision, recall and F-measure. We used the following month of project data as a test set instead of cross validation to mimic a realistic use of the recommender. We describe the project data used in the experiments (Section 4.1.1), our method of evaluation in more detail (Section 4.1.2) and the results (Section 4.1.3).

4.1.1. Projects. For the laboratory experiments, we used data from the gcc compiler project,[14] the Mylyn project,[15] and the Bugzilla project. For ease of reference, we also report in this section the results of the two projects used in tuning our approach, the Eclipse Platform project and the Firefox project.

These five projects represent a range of triage processes and domains, small to large numbers of developers, and low ($<$10 per day) to high ($>$20 per day) bug submission frequencies. Table VII shows the different dimensions for the projects. The second column categorizes the projects based on their triage process: developer-based or volunteer-based. In a developer-based triage process, the triagers are developers who typically take turns performing triage for a subsystem of the project. Projects that use a volunteer-based triage process rely on volunteers from the project community for triage; these volunteers, who may not be developers, may have less project-specific knowledge than an experienced triager. As the recommendations our approach makes our based on which developer was responsible for resolving a report, the type of triage process should not affect the quality of the recommendations.

The next column describes the application domain of the different projects. The fourth column gives an estimate[16] of the number of developers for each project based on our eight-month data sets. In other words, the number of developers for a project is the number of distinct developer labels for all the reports in our data set. Table VIII 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 IX).

4.1.2. Evaluation Method. Table X shows the number of active developers, the number of training reports, and the number of testing reports used in the evaluation. This data

[14]Gcc is a collection of compilers and can be found at www.gnu.org/software/gcc/gcc.html (verified 3/21/09).
[15]Mylyn is a task-focused UI plug-in for Eclipse and can be found at www.eclipse.org/mylyn (verified 3/21/09).
[16]As these values are based on our project heuristics, these values are estimates rather than actuals.

Table VIII. 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 IX. 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

Table X. Training and Testing Set Sizes for Evaluating Developer Recommenders

Project	Developers	Training Reports	Testing Reports
Eclipse	44	6356	152
Firefox	56	3338	64
gcc	31	2521	70
Mylyn	6	683	50
Bugzilla	11	799	52

shows that the gcc project has a similar number of active developers as the Eclipse and Firefox projects and that the Mylyn and Bugzilla projects have few active developers.

To evaluate a developer recommender, 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 developer recommender. The easiest way to determine the set of developers with implementation expertise for a report is to ask an expert in the project. As we did not have access to such experts for all the projects, we developed heuristics for each project based on information in the source revision repositories.[17] Our test sets consist of the resolved reports from the month following the date range used for the training data set for which an implementation expertise set can be determined.

At a high level, the implementation expertise set for a particular bug report is determined by mapping a report to the source revision log entries detailing which source files changed to resolve the report. Each log entry includes the id of the developer creating the log comment. From these entries, a list of developer names is constructed that represents the developers who have the expertise to resolve the report. As developers who work in the same area of the system will likely also have pertinent expertise, the heuristic uses the notion of a containing module, such as a Java package or a specific depth in the source code folder hierarchy. The heuristic compiles the implementation expertise list from the log entries of the files in the containing module. Although there are other techniques, such as the use of bug networks [Sandusky et al. 2004], that may be used to determine the implementation expertise set for a bug report, our previous work found that the source repository technique was better suited for the evaluation of a developer recommender [Anvik and Murphy 2007]. Further details about constructing the implementation lists can be found in our previous work [Anvik et al. 2006; Anvik and Murphy 2007; Anvik 2007].

[17]This evaluation technique has also been used by others [McDonald 2001; Mockus and Herbsleb 2002].

Table XI. Size of Recommendation Pool and Average Implementation Expertise List Size

	Eclipse	Firefox	gcc	Mylyn	Bugzilla
No. of Dev. in Recommendation Pool	44	56	31	6	11
Avg. Size of Implementation Expertise List	8	80	43	3	32

Table XII. Precision, Recall and F-Measure (as Percentages) 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

Using the technique of mapping of bug reports to names from the source repository has an important consequence: it over-estimates the set of developers with implementation expertise, especially for projects such as Firefox and Bugzilla where an additional mapping is needed due to the project's development process.[18] 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 over-estimation 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 developer recommender. 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 developer recommender results, as we favor precision over recall.

4.1.3. Results. Table X and Table XI provide an overview of the data used in the evaluation. The first row in Table XI reports the number of developers available to be recommended after cleansing the data in the repository as described in Sections 3.2.4 and 3.2.5. The second row in Table XI reports the average of the number of developers who may have had the likely expertise to solve reports used in the evaluation process as computed from the implementation expertise sets. To ensure we can calculate precision as accurately as possible, we did not place any restrictions on the period of time from which we mined the implementation expertise lists from the revision logs. As a result the values in the second row of Table XI are sometimes larger than the number of developers in the recommendation pool.

Table XII shows the results of applying the recommender creation process to create a developer recommender for the gcc, Mylyn and Bugzilla projects, as well as the results from tuning with Eclipse and Firefox. The values reported in Table XII are the average over the recommendations made for reports in the testing set. Figure 4 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 precision for all these projects.[19] For a developer recommender, as we are interested in choosing an appropriate developer to whom to assign the report rather than to determine all possible developers that might have appropriate expertise, we favour precision over recall.

[18]Although this over-estimation could lead to an artificial inflation of the precision values, our earlier empirical work suggests that any artificial inflation will not be significant [Anvik and Murphy 2007].
[19]Our previous work indicated that our approach did not work well for gcc [Anvik et al. 2006]. Further examination revealed the cause to be an error in our data collection for the implementation expertise sets.

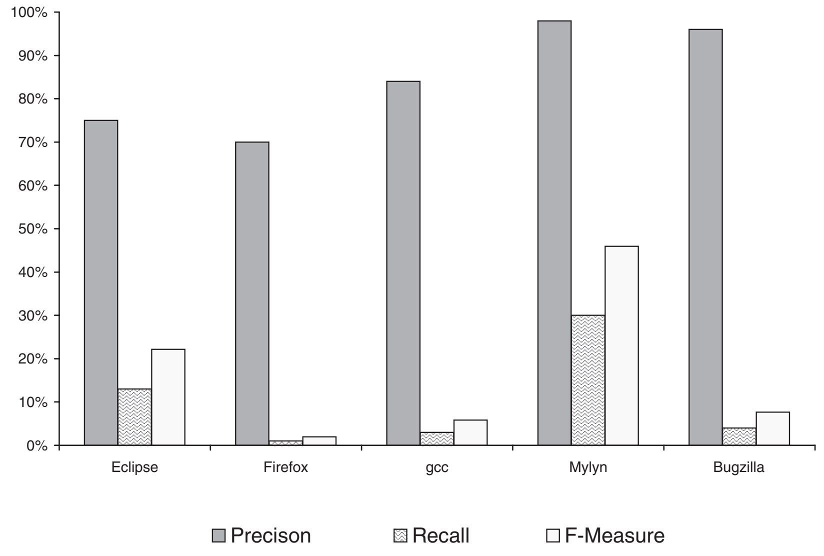


Fig. 4. Graph of the five developer recommenders for the top recommendation.

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 developer recommender 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, that has roughly three to five times as many developers from which to make a recommendation.

Compared to the other projects, the developer recommender for Mylyn has a very high recall. We believe that this is a result of our evaluation methodology. As previously mentioned, our evaluation technique is known to over-estimate the size of the implementation expertise set [Anvik and Murphy 2007]. Of all the projects, Mylyn has the smallest pool of developers who might have expertise for a particular report, therefore the over-estimation of the developers with implementation expertise for a particular report will be less for Mylyn and the error in the recall value will be less.

4.1.4. Threats to Validity. A threat to the construct validity of our laboratory experiments is our technique for determining implementation expertise. As was stated previously, the technique over-estimates the number of developers who have implementation expertise for a particular report. Although this improves the accuracy of our precision measurement, it does so at the expense of hindering our ability to accurately determine recall. For developer recommenders, as we are more interested in precision over recall, we think this threat is contained.

A similar threat to the internal validity of our laboratory experiments is the mapping of user names between the bug and source repositories. Although we made every attempt to correct the mappings, it is possible some mappings were in error.

By evaluating our approach on multiple open source projects with a variety of characteristics, we increase the likelihood that our approach applies across a variety of

Table XIII. 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

situations (i.e., external validity). It may be that our approach is not applicable for projects with different characteristics, such as closed source projects where the reports in the bug repository may follow a different life cycle.

4.1.5. Summary. The laboratory experiments were designed to answer the question of whether our approach can create recommenders that make accurate recommendations. These experiments have shown that, when applied to the problem of providing a developer recommender making one suggestion, the approach achieves reasonably high precision, ranging from 70% for Firefox to 98% for Bugzilla. These precision rates show promise for lowering the time it takes a triager, whether a novice or experienced triager, to make an assignment for a report.

4.2. A Field Study of Triage Recommenders

The results of our laboratory experiments leaves open the question of whether a human can make effective use of information recommended by our approach. To gain insight into this question, we conducted a field study with four triagers from the UI team of the Eclipse Platform project.

4.2.1. Participants. Each of these triagers had substantial knowledge about the system for which they were triaging. Table XIII provides background about this experience, showing details about the triager's reported project and triage experience, and time spent triaging reports.

4.2.2. Method. Over the course of four months, we gathered data from the triagers as they used Sibyl as part of their triage activities. Sibyl was configured with three development-oriented recommenders: a developer recommender to help in the assignment of a report, a component recommender to help with ensuring the report is assigned to the right part of a project and an interest recommender to help with ensuring appropriate developers are made aware of the report. We focus in this section on the use of the developer recommender, which was used by the triagers to make 259 assignments. This value does not reflect all assignments made for this project over the study period because the triagers needed to remember to perform the triage activities through our tool. The tool added some overhead to the triage process because of its setup as a proxy; on average, it added approximately four seconds for a triager to get a report augmented with recommendations with a worst case during the study of over an additional minute.

As the triagers used Sibyl, we gathered quantitative data about their use of the recommenders and the accuracy of the recommendations provided by Sibyl. We also gathered qualitative data via questionnaires and post-usage interviews. Figures 5 and 6 show the questions that were posed to the triagers during the use of Sibyl. The post-usage interview questions collected more general data about the triager's approach to triage and impressions of Sibyl.

4.2.3. Results. We were interested in whether or not the triagers made use of the developer recommendations. We gauge this by computing the *average accuracy* of the recommender (see Formula 4). If the triager selected one of the developers recommended

(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.

Fig. 5. 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?

Fig. 6. Questions for the usage questionnaire.

by Sibyl, we considered that a success. If the triager entered a developer name that was not recommended, we considered that a failure. We found that in practice the developer recommender had an average in-practice accuracy of 75% when presenting four names.[20] In other words, three quarters of the time the correct assignment was recommended. If one takes into consideration instances where the recommender was provided incorrect information, such as asking for developer recommendations for a report that was incorrectly filed to the UI component, then the average accuracy rises to 84%. Data from the post-usage interviews agrees with these high accuracy computations. One of the triagers felt that Sibyl's developer recommender had a high accuracy. Another triager commented that Sibyl gave good recommendations but that Sibyl would sometimes get confused by such things as stack traces in the problem description.

$$Accuracy = \frac{\#\ of\ reports\ for\ which\ a\ provided\ recommendation\ was\ used}{\#\ of\ reports\ for\ which\ recommendations\ were\ given} \tag{4}$$

From the data collected via questionnaires, the triagers found that in general recommending four developers was sufficient, although this was sometimes too few. Although the triagers indicated that there were cases where multiple recommendations were applicable, they felt that choosing between them was not difficult. Only one triager completed the usage questionnaire (Figure 6). The triager felt that the developer recommender was very useful and made him consider other alternatives that he might not otherwise have considered. The triager reported no differences in between the recommendations and his expectations. He also felt that Sibyl made it faster to do assignments.

[20]The recommender was selecting from a pool of nineteen UI developers as opposed to the forty-four developers from across the entire Eclipse platform project shown in Table X.

From the study, we had initially hoped to determine if there were time savings from the use of the recommendations. We did not see any time improvement, however the statistical power of our analysis was very low (12%) given the small number of participants, making the data very difficult to interpret.

Further details of the field study are available elsewhere [Anvik 2007].

4.2.4. 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. Similarily, all of the participants worked on the same component of the project and used a developer recommender specific to that component. Therefore, the results that the developer recommender had a high in-practice accuracy may not extend to other projects or components of the same project.

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 of 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, but we were unsuccessful in recruiting more participants.

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 accuracy results for the developer recommender. 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 later 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).

Further threats are detailed elsewhere [Anvik 2007].

4.2.5. Summary. The results of the field study provide evidence that the accuracy of the recommenders is sufficiently high to be of use to human triagers. The four experienced triagers who participated in the field study were positive about the value of the recommendations. Unfortunately, the small number of participants and the relatively small number of assignments performed by these triagers made it impossible to determine if providing the triagers recommendations results in a time savings for triaging reports.

5. ASSISTING RECOMMENDER CONFIGURATION

As explained in Section 3, the configuration of a development-oriented recommender is conceptually straightforward: six questions need to be answered. However, in practice determining the answers to these questions is challenging. For example, answering the questions of which labels are valid (see Section 3.2.5) and which algorithm to use (see Section 3.2.6) required significant experimentation for just one type of development-oriented recommender.

Some of these answers, such as which machine learning algorithm to use or which bug report features to select, are likely applicable across multiple types of development-oriented recommenders. However, most of the questions require different answers for each type of recommender. For example, how labels are determined for a component recommender (Section 6.2.1) or an interest recommender (Section 6.2.2) differ from how labels are determined for a developer recommender.

Not only do these answers vary across types of development-oriented recommenders, the answers vary across instances of the same type of recommender. In creating a developer recommender, the heuristics used for labeling Eclipse reports are not the same as those used when creating a developer recommender for the Firefox project. Creating such heuristics can be a time-consuming process. In our work, we found that to create and tune the heuristics for a new project took approximately a half a day to a full day depending on the complexity of the heuristics.

To answer the recommender configuration questions, a project member would typically need information from the bug repository. Even if the project has a prescribed process for using the repository, the fact that the data is manipulated by humans means it can vary from what is expected. For instance, after fixing a simple bug some developers may never change the state to the ASSIGNED state, moving the bugs instead directly from the NEW state to the RESOLVED state, while other developers rigorously follow the project's practice of first assigning the report to themselves before marking the report RESOLVED. If the labeling heuristics expect reports to always be assigned to the developer who fixed the problem, then there will be set of reports that will not be used for training as they cannot be labeled, and the recommender will not have as complete a knowledge of the developers expertise as possible. This will likely limit the effectiveness of the project's developer recommender. By considering the actual data being used to train the recommender, the project member can help ensure the recommender is trained with best data possible.

Moreover, if there are any significant changes in the kinds of data input to the repository, the recommender may need to be reconfigured. For example, the Eclipse project deprecated two of their life cycle states, REMIND and LATER, requiring a reconfiguration of the project-specific labeling heuristics and which reports to use for the project's developer recommender.

The initial configuration and ongoing adjustments for creating a recommender represents a cost to the project. Even if a project decides to use only a single type of recommender, this cost can be significant. For example, the Mozilla project has four different web browser projects, Firefox, SeaMonkey,[21] Camino,[22] and Minimo.[23] For each of these projects there is a subset of developers who work exclusively on one of these browser projects, and a subset of developers who also overlap in the Core project used by all the browser projects. If the Mozilla project decides to incorporate a developer recommender into its development process, the existence of these developer subsets requires that a different developer recommender be created for each subproject. At the limit, the Mozilla project, which has thirty-five subprojects, would have to create and maintain thirty-five different developer recommender configurations. This could pose a substantial cost to the project.

The recommender configuration problem is further compounded as a project incorporates more recommenders into its development process. Assuming that the Mozilla project decides to use three different types of development-oriented recommenders (e.g., developer, component, and interest) and half of the projects require a unique recommender configurations, then the project must create and maintaining fifty-one different recommender configurations.

To make the use of development-oriented recommenders practical, these costs need to be reduced. As it is unlikely that we can reduce the number of different configurations that will need to be created and maintained by a project, we seek to lower the costs associated with each individual configuration. As before, when describing the overall approach, we again focus on the configuration of a developer recommender. The first technique assists in determining the project-specific heuristics for report labeling. The other technique assists in determining the developer activity threshold for the project.

[21]http://www.seamonkey-project.org, verified 4/22/08.
[22]http://caminobrowser.org, verified 4/22/08.
[23]http://www.mozilla.org/projects/minimo, verified 4/22/08.

Table XIV. Legend for Bug Report Life Cycle States

U	UNCONFIRMED
N	NEW
A	ASSIGNED
D	DUPLICATE
I	INVALID
F	RESOLVED-FIXED
V	VERIFIED-FIXED

As we will show, by assisting with these two decisions, we also assist with determining which reports to use for training the recommender. We discuss later how these techniques can be extended for use with other types of recommenders (see Sections 6.3).

This section continues by presenting the two techniques for assisting development-oriented recommender configuration. The techniques reduce the configuration costs in a similar manner to that of the recommenders themselves. The techniques provide a user with data so that she does not have to collect the information, instead she makes configuration decisions based on her project-specific knowledge. We conclude the section with an evaluation of developer recommenders created using the assisted configuration techniques to show that the recommenders are "sufficiently good."

5.1. Specifying the Labeling Heuristics

In our work from Section 3.2.4, when we configured a developer assignment recommender for a project, we had to use project documentation, discussions with project personnel and inspection of (often many) bug reports in the repository to determine how to identify from a report which developer should be labeled as resolving the report. This process was time consuming. Although this time could be considered an investment to be paid back by the savings of using recommenders for triage, if there were any changes to the development process the data collection process would likely have to be repeated. To reduce this time and effort, we describe a technique to automatically characterize a sample of repository reports and aid the user in identifying which reports to label and how to label the reports.

5.1.1. Grouping Reports by Life Cycle. To create effective heuristics for a project, the reports in the repository need to be categorized. For each report category, a heuristic is applied to provide the labels. Since for a developer recommender, the labeling of reports depends upon actions performed to the bugs, we base the categorization on the life cycle history of a bug report. We use this form of grouping into categories because reports that have reached certain stages in the bug report life cycle provide different information. For example, reports that are in a NEW state are not likely to provide information about who is assigned which types of reports, whereas a report that has reached a RESOLVED-FIXED state will most likely have such information.

We determine the life cycle path of each report from the history log of the report. We represent a life cycle path as a string in which each character represents a state that the report has passed through (see Table XIV for a mapping of characters to states). For example, the life cycle path NAF represents a report that starts in the NEW state, moves to the ASSIGNED state and concludes in the RESOLVED-FIX.

To group the reports based on these strings, we derive regular expressions to represent similar life cycle paths. We refer to these regular expressions and the reports that map to them as *path groups*. We use a heuristic approach derived from observations of common path patterns found in a variety of projects to create the path groups as determining a regular expression from a set of examples is known to be a hard problem.[24]

The path groups consist of four forms of regular expressions:

(1) a string representing a single path,
(2) expressions capturing a common cycle,
(3) 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 forms.

These forms increase in their level of generality. These different levels of generality can be useful in specifying configuration parameters as will be evident later in this discussion.

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.

We observed a common occurrence with projects that have two states for representing a new report. 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 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 form consists of various combinations of the second and third forms, such as (U)?(NA)+F(V)?. This path expression 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.

There are two ways that path groups can be formed from a repository of reports. First, all the reports in the repository can be examined to create the path groups. However, this approach can place a significant burden on the project's infrastructure. Also, as the set of reports can span years of development, this approach risks collecting data about a project's obsolete development process. Alternatively, just those reports in the recent past can be used to form the path groups. In this approach, we randomly sample reports at intervals of one, two, four, and eight weeks in the recent past. These choices of intervals were used to experimentally determine the best sample and were chosen to favor the most recent information so as allow the capture of changes in the project's development process as soon as possible. A sample for a single interval consists of a maximum of twenty-five reports, resulting in a maximum total sample of 100 reports.

[24]The survey paper by Sakakibara [1997] on grammatical inference lists several bodies of work that provide computationally hardness results for this process.

Table XV. Number of Reports in Data Sets Covered by the Path Groups

Project	Reports	Path Groups	Report Coverage
Eclipse	10297	25	77%
Firefox	9047	38	72%
gcc	3841	24	73%
Mylyn	842	12	82%
Bugzilla	1566	21	76%

We found that increasing the sample size did not cause a significant change in the relative frequency of path groups.

From the path groups created from the sampled reports, we remove those that contain only one or two sampled reports as these are deemed to be too specific. For the five systems, we have tested (see Section 4), this resulted in a range of 12 to 38 expressions, a tractable number for a user to scan to identify path groups of interest.

Table XV shows the number of reports from our five data sets, how many path groups were determined from the sample, and the percentage of reports that are covered by at least one path group in the filtered set. The path group coverage ranges from 72% to 82% of the reports in the data sets.

5.1.2. Determining Data Sources. If it was possible to specify one rule to describe how to label all bug reports from a repository prior to feeding the reports to the recommender, configuration would not be overly difficult. Unfortunately, a set of labeling heuristics is typically needed to cover all of the different kinds of reports of interest. For example, in the developer recommender we previously created for Firefox, we used nine heuristics that differentiated between such cases as when there was an approved patch associated with a bug—the presence of an approved patch supersedes any information about who was assigned the report because the patch submitter has more expertise about how to fix those kinds of bugs.

To make it tractable to specify the labeling heuristics, we use an approach of automatically extracting data sources from reports that belong to a path group. For a developer recommender, a suitable data source is a field of the report or an event in the report's activity log that can provide information about project developers. The specification of suitable data sources is given by the creator of the data extraction module for configuring a particular recommender.

In this technique, a user specifies which data sources should be used to label reports from a particular path group. Examples of data sources include the value of a bug's assigned-to field or the person who attached a patch to a report. The potential data sources for reports of a path group are mined from occurrences of user names in the fields and history log of the reports. An additional case is the link between a duplicate report and the report it duplicates that allows the fields and history log of the duplicated report to be used for labeling. For example, the mining of a set of reports might result in three data sources: an attachment-added event in the log, a resolved-fixed event in the log, and the value of the assigned-to field. Not all data sources may be available for each member of a path group; as a result, multiple data sources may be specified for a single path group.

Table XVI shows examples of path groups from the Firefox project with extracted data sources and the data sources selected to create the labelling heuristic. The first example shows the path group NA(F)? and the identified data sources for reports in this group. Note that not all reports in this path group will have all of the listed data sources. In cases such as this, where multiple data sources are needed to label a single path group, an ordering for the selected data sources is specified. The last column of

Table XVI. Path Groups and Potential Data Sources from the Firefox Project

Path Group	Data Sources Detected	Data Sources Specified
NA(F)? (reports that begin as NEW, transition to ASSIGNED and possibly transition to RESOLVED-FIXED state)	Assigned To Reporter Made Assignment Marked Fixed Set Status QA Contact In cc: List Submitted Last Patch	(1) Submitted Last Patch (2) Assigned To
UD (reports that begin as UNCONFIRMED and transition to an INVALID state)	Assigned To Reporter Made Assignment Label of Duplicate Report QA Contact Added Attachment Set Status Resolved By In cc: List	Label of Duplicate Report
UI (reports that begin as UNCONFIRMED and transition to a DUPLICATE state)	Assigned To Reporter Set Status QA Contact Resolved By In cc: List	(No data source selected)

the row shows that for labeling reports in this path group the “Submitted Last Patch” log event is first used. If this data source does not exist for the particular report, the value of the “Assigned To” field is then used.

The second example shows the path group UD. One of the identified data sources indicates a “duplicate report” link. Selecting this data source indicates that reports in this path group should follow their duplicate report link and apply the appropriate heuristic to the duplicated report to determine this report’s label.

The final example shows the UI path group. This path group indicates reports that have been marked as INVALID by the triager and are never assigned to a developer. As these reports do not have sufficient information for labeling a report for training a developer recommender, no data source is selected.

5.2. Selecting the Labels

Once reports can be labeled, it is also necessary as part of the configuration process to determine which labels are valid for recommendation. As discussed in Section 3.2.5, not all developers listed in the repository are necessarily currently active on the project. We thus need to remove these developers from consideration, which corresponds to selecting the labels for use in the recommender. There are two possible ways to select labels. First, the user could use their own project knowledge to provide a list of developers that may be used to label the reports. Second, a threshold can be derived for determining which labels to consider valid based on a developer’s activity in the repository. As explained in Section 3.2.5, we have focused on the latter approach to enable more graceful evolution with changes to repository data.

We created a graph of report resolution by developer over the most recent three months using the labeling heuristics. The x-axis represents individual developers and the y-axis represents the number of resolved reports (Figure 7); the developers are ordered from those resolving the most reports to those resolving the least. Using an Anderson Darling for the five test projects we have considered, we found that a Pareto distribution best modeled the report resolution curves for the projects. Figures 8 and 9 show the developer resolution graphs from our Eclipse and Firefox data sets

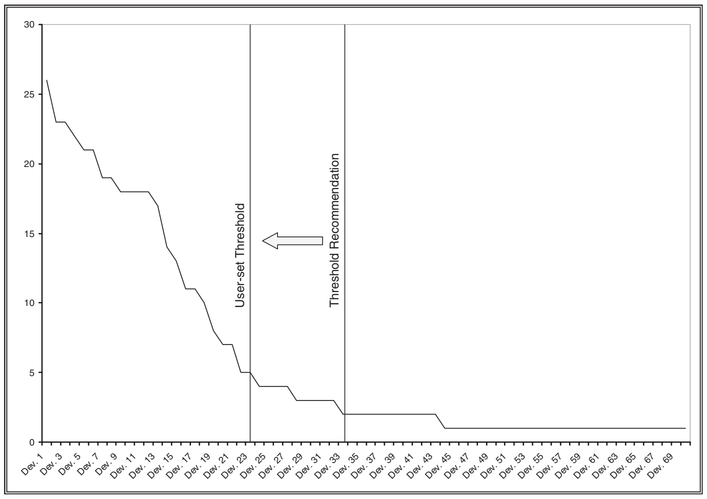


Fig. 7. Example of setting the reports-resolved threshold for selecting the active developer set.

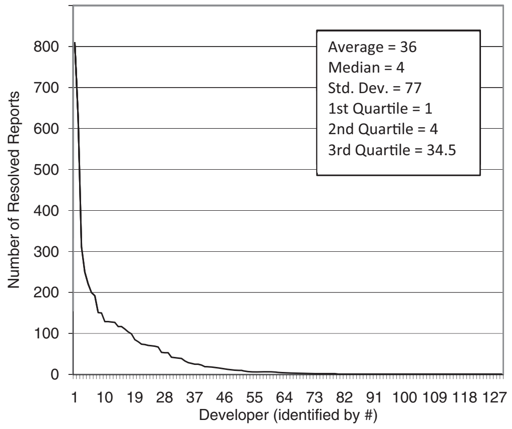


Fig. 8. Developer resolution graph for Eclipse.

respectively. Similar to Figure 7, the x-axis represents individual developers and the y-axis is the number of reports resolved by that developer. An initial starting point for choosing a threshold for a cut-off activity level is then the median of the Pareto distribution created from a project's report resolution curve. This value could be altered by a user choosing the threshold as shown in Figure 7 where the user has slid the cut-off from two to five based on project knowledge.

5.3. Choosing the Reports

Recall from Section 3.2.1 that only reports that can be labeled are used for training a development-oriented recommender. As the user selects specific heuristics for different bug report life cycle path groups, she also specifies which report to use for training. Similarly, as the user selects which labels are valid, she is selecting which reports to use as reports with invalid labels are not used in the recommender training process.

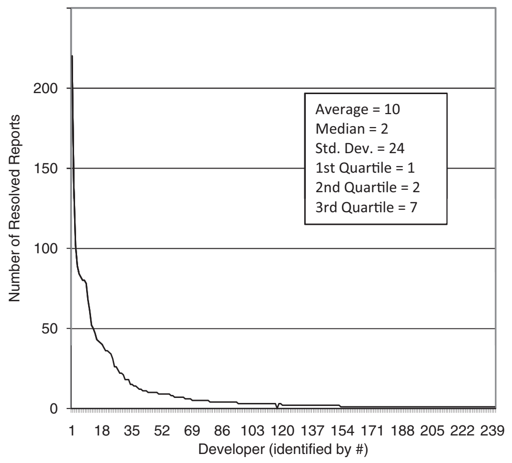


Fig. 9. Developer resolution graph for Firefox.

Table XVII. Date Ranges for 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

5.4. An Evaluation of the Assisted Configuration Techniques

For the presented configuration techniques to be useful, we need to show that the recommenders produced using these techniques are “sufficiently good.” We assess “sufficiently good” by comparing the precision and recall values of recommenders created using the configuration techniques with the values for recommenders created using our previous hand-configured approach from Section 3. We consider a “sufficiently good” result one that is within 15% of the precision of the previously created recommenders. We chose 15% because a decrease of this amount in precision would still result in a recommender that provides a correct assignment recommendation better than half of the time. From interviews with industrial developers who used the approach, we know that accuracy rates of 75% were useful (see Section 4.2); we believe a decrease of 15% would still render the recommenders effective. We chose to limit our evaluation to a comparison against previous results as a human evaluation would be unwarranted if sufficiently good results cannot be achieved.

As before, we use precision, recall, and F-measure to evaluate the effectiveness of the created recommenders (Formulas 1 through 3 of Section 3.1). Recall that although we present both precision and recall 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 false positives. Our evaluation considers a recommendation set of size one. Table XVII describes the data used in the evaluation; Table XVIII reports the values of these metrics for the five projects using our original approach.

One challenge in evaluating recommenders is that the many decisions made in configuring a recommender inter-relate. To account for this interplay between the decisions, we attempt to isolate the two decisions supported by our configuration techniques,

Table XVIII. Overview of Recommenders Created Using the Recommender Creation Process

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

Table XIX. Active Developer Thresholds

	Median		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

the heuristics used and the threshold for selecting active developers, by evaluating combinations of two different data points for each decision.

For the heuristics decision, we wanted to investigate how sensitive the results were to the number of path groups for which heuristics were specified. We ordered the path groups by the percentage of sample reports described by the path group's regular expression. We then compared the recommenders that result from specifying heuristics for the top five and top ten path groups ranked by their coverage of the sample reports. In specifying the heuristics, we used domain knowledge that a project member would likely have, which we acquired from constructing the heuristics from Section 3.2.4, to make the obvious choices for each path group's data source(s). In one case, this resulted in not assigning data sources for some of the top ten path groups of the Firefox project, because they represented 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.

For the active developer threshold decision, we wanted to gauge how sensitive the recommender configuration process is to the value chosen. Table XIX shows the median values from applying the Pareto distribution to a project's data and a point we judged to be where the resolution curve started to flatten. We investigated these two points as we felt that the median of the Pareto curve was lower than was likely reasonable; we believe the median is typically low because there are often a large number of developers who only resolve one report, leading to a long tail in the Pareto curve.

Further details of the individual recommender configurations, such as the specific heuristics used, are found in Appendix B.

5.4.1. Results. Tables XX and XXI report the results of the configured recommenders for the various combinations of these two decisions. The first column of the two tables list the five projects used to evaluate the assisted configuration techniques. Tables XX shows the results for using the heuristics for the top five path groups and XXI shows the results for using the top ten path groups. The tables are divided into two sections corresponding to the threshold used for determining the active set of developers; either the suggested threshold or the threshold we chose based on visually inspecting the graph. For each case, the table also shows the amount of data used to train a recommender, the threshold value, and the precision and recall values for the recommender. Appendix B provides further details about the underlying data used in this evaluation.

Table XX. Configuring an Assignment Recommender Using Heuristics for the Top Five Path Groups

	Data Size	Threshold (Median)	P/R/F (%)	Data Size	Threshold (User)	P/R/F (%)
Eclipse	1722	2	62/10/17	1650	5	63/10/17
Firefox	1270	2	64/1/2	1025	5	68/1/2
gcc	329	2	70/3/6	293	5	70/3/6
Mylyn	560	3	98/30/46	556	10	98/30/46
Bugzilla	379	2	100/6/11	342	5	100/6/11

Table XXI. Configuring an Assignment Recommender Using Heuristics for the Top Ten Path Groups

	Data Size	Threshold (Median)	P/R/F (%)	Data Size	Threshold (User)	P/R/F (%)
Eclipse	2907	2	72/13/22	2861	5	72/13/22
Firefox	1270	2	63/1/2	1025	5	67/1/2
gcc	629	2	83/3/6	518	5	82/3/6
Mylyn	560	3	98/30/46	556	10	98/30/46
Bugzilla	406	2	100/6/11	374	5	100/6/11

Overall, the data in Tables XX and XXI demonstrate that the configured recommenders are tolerant to various choices made during configuration. For all cases, the configured recommenders are within our 15% tolerance for precision achieved using a much more time-consuming configuration approach and significantly more data from the repository (see Table XVIII).

Specifically, the data in Table XX shows that the use of the top five heuristics is sufficient to produce a recommender with precision in the range of 62%–100% depending on the threshold used. However, as would be expected, the use of even more heuristics (see Table XXI) generally produces a recommender with even better precision. For example, the Eclipse Platform recommender's precision improved from 62% to 72%. That the precision would improve with the use of more heuristics makes sense as more heuristics provides for a larger set of reports for training. For example, the number of reports used for training the gcc recommender increased from 329 reports to 629 reports when using ten heuristics, and the recommender's precision improved from 70% to 82%.

Tables XX and XXI also show 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 biggest difference is for the Firefox project where changing the activity threshold from two reports to five reports resulted in a 4% improvement in precision. For the rest of projects, changing the threshold changed the precision at most 1%.

Through this experimentation, we were also able to gauge how long configuration takes. Once the sample data has been acquired from a repository, the time to compute and present the path group, data source, and developer resolution information is approximately two minutes. If the amount of data to use for training the recommender has been determined and acquired, then creating a newly configured assignment recommender also takes a short amount of time. For example, training a recommender for the Eclipse Platform using 2300 reports takes less than two minutes. We believe these times are completely reasonable to allow a user to iteratively tune a recommender using the assisted configuration techniques.

6. EXTENDING RECOMMENDER CREATION AND CONFIGURATION

In this section, we discuss the use of additional process information to improve the accuracy of a created recommender. We also discuss how the recommender creation process can be used to create other types of development-oriented recommenders and

Table XXII. Precision and Recall for a Component-Based Developer Recommender for the Five Projects as Percentages

# Recommendations	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 XXIII. Precision, Recall and F-Measure for a Developer Recommender for the Five Projects as Percentages

# Recommendations	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

how the configuration techniques introduced in Section 5 can be used to assist in the configuration of these additional recommenders.

6.1. Using Other Development Process Information to Create a Recommender

The recommender creation approach presented in Section 3 takes into consideration only one type of development process information. For a developer recommender, this means that the approach only considered information about how reports were assigned to individual developers. However, we have observed that the development teams for the Eclipse and Firefox projects are 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 [Di Lucca et al. 2002; Mockus et al. 2002]. Could a more accurate recommender be created if we took into consideration more information about the development process? In other words, could a more accurate developer recommender be created if we incorporate the knowledge that developers are structured into teams based on the product component?

To construct such a recommender, we first group the reports by reported component and then the groups of reports are used to train a separate recommender for each component. We refer to a developer recommender formed in this way as a component-based developer recommender. 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 [Anvik et al. 2006]. 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 [2007] described this phenomenon as emergent teams. For a new report, the recommender uses the appropriate component-based recommender based on the value of the component field in the report to make the recommendations.

Table XXII shows the analytical results of using this process for the five projects to create developer recommender; these results are based on the same data set as used in Section 4. To ease comparison with the previous results, Table XXIII reiterates the results of the analytical evaluation reported in the earlier section. From Tables XXII and XXIII, we see that using the process information substantially improved the precision of the recommender for Eclipse across different recommendation list sizes, 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 [Anvik et al. 2006], however data from other work seems to indicate that this is not the case [Minto and Murphy 2007].

Table XXIV. Recall for the Five Component Recommenders

	Eclipse	Firefox	gcc	Mylyn	Bugzilla
Components	18	34	33	11	16
1 Prediction	66%	57%	55%	53%	45%
2 Predictions	85%	72%	68%	66%	62%
3 Predictions	92%	79%	76%	83%	74%

This result may also be a consequence of emergent teams being more prevalent for the other projects.

6.2. Creating Recommenders for Other Development-Oriented Decisions

The approach described earlier to create a developer recommender can also be used to create recommenders for other development-oriented decisions [Anvik 2007]. In this section, we present an overview of how the recommender creation process has been used to create two other types of recommenders. The first recommender is a triage recommender that suggests the component against which to file a report. The second recommender suggests which other project members may be interested in being kept aware of changes to the report. Results from experiments against historical data for these two types of recommenders are given below. Empirical evaluation results are available elsewhere [Anvik 2007].

6.2.1. 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[25] 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, likely 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.

Most of the answers to the questions for a developer recommender can be used to create a component recommender. The two answers that differ is how report labels are determined and how valid labels are determined. As opposed to using project-specific heuristics for labeling the reports, the value of the component field is used instead. Also, whereas the labels for a developer recommender required pruning, all labels are considered valid for a component recommender.

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 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.

Table XXIV shows the results for using our approach to create a component recommender for each of five different projects. The row labeled "Components" shows the number of components for the project. We see from the table that if we use the top recommendation (the row labeled "1" in Table XXIV), 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, which is likely a effective range for the recommendations to be useful.

[25]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 XXV. Precision, Recall and F-Measure for the Five Interest Recommenders

	Eclipse	Firefox	gcc	Mylyn	Bugzilla
Names	58	108	21	5	17
Predictions	Precision/Recall/F-measure (as percentages)				
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

6.2.2. 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: field 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, she may be interested because she is working on a bug for which this bug is blocking her 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.[26] An interest recommender assists the triager in deciding which other project members should be aware of a report.

In creating an interest recommender, the answers to three questions differ from those for a developer recommender. First, labels are extracted from the list in the cc: field. Second, only names that occur in fifteen or more reports are considered valid. Lastly, reports are collected from three months as opposed to eight months. The short time frame is because there are often multiple individuals that are interested in a report, and for training a copy of the report is made for each of the names. This results in the creation of a lot more data than is created by for a developer recommender for the same time period.

Table XXV shows the results for the interest recommender of the five projects for up to seven recommendations. The row labeled “Names” shows the number of names from the cc: field that met the threshold criteria. We chose to investigate the presentation of seven interest recommendations, as we want to determine how many recommendations need to be presented for the recommender to have a good recall. As with the component recommender, we favor recall over precision for an interest recommender.

The data in Table XXV shows that the interest recommenders achieve recall levels of between 46% to 72% for seven recommendations (or five in the case of Mylyn). In other words, an interest recommender 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, 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.

6.3. Assisted Configuration of Other Recommenders

We believe our assisted configuration approach generalizes beyond configuration of a developer recommender. To discuss this generality, we describe how the assisted configuration techniques can be used to configure component and interest recommenders.

[26]Personal communication with Mozilla developer, 3/1/05.

For a component recommender, there are two changes. 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 developer recommender 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.

As with the component recommender, for an interest recommender there are two changes to how the techniques are applied. First, we need to change the data sources 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 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.

7. DISCUSSION

In this section, we discuss whether these approaches also apply to repository-oriented decisions and describe some potential directions for future work.

7.1. Applicability of Approach to Repository-Oriented Decisions

Our focus on development-oriented decision comes from the following observation: many of the repository-oriented decisions fall into one of the two situations for which our approach is not applicable. The first situation is the trivial case: a recommender is not appropriate for the decision. For example, determining if a described problem is reproducible is not an appropriate application of a recommender. Creating such a recommender would be similar to creating a test suite for the application.

The second situation is when there are multiple categories but very little data to distinguish between them. For example, our overall approach is not applicable for creating a duplicate detector.[27] Recall from Section 2, that a duplicate detector suggests whether or not a problem has been previously reported or a feature has been previously requested. In this situation, our approach is not applicable as there are multiple categories, the groups of duplicates reports, with little data for each group, as the majority of the reports are either unique or contain a small number of reports. In addition, the Support Vector Machines algorithm that we use to create the recommender does not provide a means for tracing back to the report to verify if the new report is truly a duplicate. Although this limitation might be resolved by the use of another algorithm, such as the nearest neighbour algorithm, such a replacement would likely be at the cost of creating a less precise recommender.

Our approach is applicable for situations where there are more than one category and there is a reasonable amount of data for each category. An example of a repository-oriented decision where our approach could be applied is to create a relevance recommender. A relevance recommender provides a suggestion for the answer to the question "Will the project fix this problem or implement this feature?". Our approach could be used to create this type of recommender by labeling each report either "Will fix" or "Won't fix" (or "Invalid"[28] which would remove the report from the training set). We leave the evaluation of such a recommender as future work.

[27]Although our overall approach may not be applicable, individual aspects of the approach may. For instance, a vector representation of a bug report can be helpful in building a duplicate detector (e.g., Runeson et al. [2007] and Hiew [2006]).

[28]Examples of reports that would be in this category are reports marked WORKSFORME or INVALID.

7.2. Future Directions

The approaches described in this article make a number of choices in the design space for recommender creation and configuration that warrant future experimentation. As one example, the approach we introduce performs minimal processing on the text in bug reports prior to applying the machine learning technique. Additional processing, such as special recognition and processing of stack traces present in some reports, may improve the performance of a recommender. As another example, we have chosen to recommend only developers who have resolved a number of reports for the project; improvements to the approach may enable the recommendation of developers who have recently joined the project but who are rapidly gaining the appropriate expertise to resolve particular kinds of reports.

Another direction for future work is to determine the appropriate number and characteristics of training reports required to produce a useful recommender with sufficient precision and recall values. Experimentation in this direction would enable the creation of prescriptive guidelines about when a development-oriented recommender might be applied successfully to a project. We have also not considered incremental techniques to update an existing recommender as bug reports are resolved for a project.

Additional work is also needed to better quantify the savings possible using an assisted configuration approach. Gathering this information likely requires longitudinal use of such an approach on a small number of projects.

8. CONCLUSION

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: repository-oriented decisions and development-oriented decisions. Effort spent making these decisions uses resources that might better be spent improving the product rather than managing the development process.

This article presented a machine learning-based approach to creating recommenders that assist with development-oriented decisions. We report on the use of this approach to create three different kinds of development-oriented recommenders: a developer recommender that suggests which developers might fix a report, a component recommender that suggests to which product component a report might pertain, and an interest recommender that suggests which developers on the project might be interested in following the report. We studied the results of applying this approach to historical information from five open source projects and found that the approach can produce recommenders with sufficiently appropriate suggestions to likely be useful. We also report on a small field study with four industrial triagers who used a developer recommender created with the approach on their own project. Overall, these triagers found the approach provided accurate recommendations; we computed an in-practice accuracy for this recommender of 75%.

Creating useful development-oriented recommenders is not a simple process of taking a set of reports and feeding them to an automatic process. Instead, a recommender must be configured based on knowledge of how a project encodes information into the reports in a bug repository and the reports used to train the machine learning algorithm must be pre-processed to reduce noise, amongst other decisions. In this article, we have outlined the many decisions that must be made and we have shown how these decisions may subtly change depending on the kind of recommender that is being created.

To help reduce the configuration effort and improve the ability for a development-oriented recommender to be used in a particular project, we introduced two techniques for assisted configuration. One technique helps in summarizing data about the reports in a bug repository using path groups as a means for selecting appropriate labels for the machine learning algorithm. The second technique helps in summarizing the activity levels of developers interacting with reports as a means of determining which labels for the reports should be considered valid. We showed that these techniques can be applied with an appropriate, not substantial, effort. Although the accuracy of the created recommender degrades, the accuracy still falls above an acceptable threshold. As development-oriented decision recommenders become more common in software development processes, assisted configuration techniques such as those presented will increase in importance to allow projects to efficiently create and effectively maintain these recommenders.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors are grateful to Cristina Conati, Prem Devanbu, and anonymous reviewers for valuable comments on different versions of this article.

REFERENCES

AHA, D. W., KIBLER, D., AND ALBERT, M. K. 1991. Instance-based learning algorithms. *Mach. Learn. 6,* 1, 37–66.

ANVIK, J., HIEW, L., AND MURPHY, G. C. 2006. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, 361–370.

ANVIK, J. AND MURPHY, G. C. 2007. Determining implementation expertise from bug reports. In *Proceedings of the 4th International Workshop on Mining Software Repositories*. IEEE Computer Society, 9–16.

ANVIK, J. K. 2007. Assisting bug report triage through recommendation. Ph.D. dissertation, University of British Columbia.

BAEZA-YATES, R. A. AND RIBEIRO-NETO, B. A. 1999. *Modern Information Retrieval*. ACM.

CANFORA, G. AND CERULO, L. 2006. Supporting change request assignment in open source development. In *Proceedings of the 21st ACM Symposium on Applied Computing*. ACM, 1767–1772.

CARSTENSEN, P. H. AND SORENSEN, C. 1995. Let's talk about bugs! *Scand. J. Inf. Syst. 7,* 1, 33–54.

CROWSTON, K., HOWISON, J., AND ANNABI, H. 2006. Information systems success in free and open source software development: theory and measures. *Softw. Proc. Improve. Pract. 11,* 2, 123–148.

ČUBRANIĆ, D. AND MURPHY, G. C. 2004. Automatic bug triage using text classification. In *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering*. 92–97.

DE SOUZA, C. R. B., REDMILES, D., MARK, G., PENIX, J., AND SIERHUIS, M. 2003. Management of interdependencies in collaborative software development. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'03)*. IEEE Computer Society Press, 294–303.

DEMPSTER, A., LAIRD, N., AND RUBIN, D. 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Stat. Soc. 39,* 1, 1–38.

DI LUCCA, G. A., PENTA, M. D., AND GRADARA, S. 2002. An approach to classify software maintenance requests. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. IEEE Computer Society Press, 93–102.

GUNN, S. R. 1998. Support Vector Machines for classification and regression. Tech. rep., Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, University of Southampton.

HIEW, L. 2006. Assisted detection of duplicate bug reports. M.S. dissertation, University of British Columbia.

HOOIMEIJER, P. AND WEIMER, W. 2007. Modeling bug report quality. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 34–43.

JOHN, G. H. AND LANGLEY, P. 1995. Estimating continous distributions in Bayesian classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*. Morgan-Kaufmann, 338–345.

MCDONALD, D. W. 2001. Evaluating expertise recommendations. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*. ACM, 214–223.

MINTO, S. AND MURPHY, G. C. 2007. Recommending emergent teams. In *Proceedings of 4th International Workshop on Mining Software Repositories*. IEEE Computer Society Press, 33–40.

MITCHELL, T. M. 1997. *Machine Learning*. WCB/McGraw-Hill.

MOCKUS, A., FIELDING, R. T., AND HERBSLEB, J. D. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Meth. 11,* 3, 309–346.

MOCKUS, A. AND HERBSLEB, J. D. 2002. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, 503–512.

MOCKUS, A. AND VOTTA, L. G. 2000. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. IEEE Computer Society Press, 120.

QUINLAN, R. 1993. *C4.5: Programs for Machine Learning*. Morgan-Kaufmann.

RAYMOND, E. S. 1999. The cathedral and the bazaar. *Knowl. Tech. Policy 12,* 3, 23–49.

REIS, C. R. AND DE MATTOS FORTES, R. P. 2002. An overview of the software engineering process and tools in the Mozilla project. In *Proceedings of the Open Source Software Development Workshop*. 155–175.

RENNIE, J. D. M., SHIH, L., TEEVAN, J., AND KARGER, D. R. 2003. Tackling the poor assumptions of Naïve Bayes classifiers. In *Proceedings of 20th International Conference on Machine Learning*. AAAI Press, 616–623.

RUNESON, P., ALEXANDERSSON, M., AND NYHOLM, O. 2007. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society Press, 499–510.

SAKAKIBARA, Y. 1997. Recent advances of grammatical inference. *Theoret. Comput. Sci. 185,* 1, 15–45.

SANDUSKY, R. J. AND GASSER, L. 2005. Negotiation and the coordination of information and activity in distributed software problem management. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'05)*. ACM, 187–196.

SANDUSKY, R. J., GASSER, L., AND RIPOCHE, G. 2004. Bug report networks: Varieties, strategies, and impacts in a F/OSS development community. In *Proceedings of ICSE Workshop on Mining Software Repositories*. 80–84.

SEBASTIANI, F. 2002. Machine learning in automated text categorization. *ACM Comput. Surv. 34,* 1, 1–47.

WANG, X., ZHANG, L., XIE, T., ANVIK, J., AND SUN, J. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, 461–470.

WEISS C., PREMRAJ, R., ZIMMERMANN, T., AND ZELLER, A. 2007. How long will it take to fix this bug? In *Proceedings of 4th International Workshop on Mining Software Repositories (MSR'07)*. IEEE Computer Society Press, 1–8.

WITTEN, I. H. AND FRANK, E. 2000. *Data Mining: Practical Machine Learning Tools with Java Implementations*. Morgan-Kaufmann.

Received May 2008; revised March 2009 and June 2009; accepted June 2009