Assisted Detection of Duplicate Bug Reports

by

Lyndon Hiew

B.Sc., The University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

May 2006

Abstract

Duplicate bug reports, reports which describe problems or enhancements for which there is already a report in a bug repository, consume time of bug triagers and software developers that might better be spent working on reports that describe unique requests. For many open source projects, the number of duplicate reports represents a significant percentage of the repository, numbering in the thousands of reports for many projects. In this thesis, we introduce an approach to suggest potential duplicate bug reports to a bug triager who is processing a new report. We tested our approach on four popular open source projects, achieving the best precision and recall rates of 29% and 50% respectively on reports from the popular Firefox open source project. We report on a user study in which we investigated whether our approach can help novice bug triagers process reports from the Firefox repository. Despite the relatively low precision and recall rates of our approach, we found that its use does increase the duplicate detection accuracy of novice bug triagers, while significantly reducing the number of searches they perform and slightly reducing the time they spend on each report.

Contents

List of Tables

List of Figures

Acknowledgements

First I would like to thank my supervisor, Gail Murphy, for helping me throughout my research. She always had time to discuss things and gave priority to reviewing my documents in a timely manner. Gail was flexible with the different directions I took and gave me invaluable advice on the next step to take. When I felt lost and did not know where my research was going; she help me see the overall plan and helped me remain grounded. It was a great opportunity to work with Gail, who is an amazing supervisor and person.

I'd also like to thank John Anvik and Brian de Alwis for their helpful advice on designing the study and the many pointers they gave me regarding my research. They were always willing to help out with any problems I had. I would also like to thank Mik Kersten for allowing me to use Mylar in the study for data collection.

Finally I'd like to thank my family for supporting my educational goals, even though they did not fully understand what I was doing and why I embarked on this path.

Chapter 1

Introduction

The reports submitted to a bug repository of an open source software development project help drive the work performed on the project. Reports that are submitted to the repository are assigned to a software developer who then determines the order in which to work on assigned reports based upon such information as a report's priority.

In practice, the process is not nearly this simple. For example, because reports can be submitted simultaneously by both users and developers, not all of the reports that enter the system are described in sufficient detail to become a work item. Other reports describe problems that can not be reproduced. To cope with the volume of less-than perfect reports, open source projects use bug triagers, people who help filter the reports down to those representing real issues and who help in the assignment of reports to developers [13].

One of the filtering steps a bug triager performs is to attempt to identify if a new report is a duplicate of an existing report. In other words, does the new report represent a problem or modification that has already been entered into the repository. When this duplicate detection process is performed well, it can significantly decrease the number of reports that must be handled downstream by developers. For instance, as of February 1, 2006, duplicate reports made up 36% of all reports in the Firefox repository,[1] 17% of Eclipse Platform reports,[2] 14% of Apache 2.0 reports[3] and 13% of Fedora Core reports.[4] These percentages correspond to large numbers of reports: 14159 for Firefox, 9181 for the Eclipse platform, 422 for Apache 2.0 and 4792 for Fedora Core.

To determine duplicate reports, a triager must currently rely either on their knowledge of the bug repository to know that they have seen the report before, or they must perform a series of searches on the repository to try to find potential

[1] `https://bugzilla.mozilla.org/`, verified 02/01/06
[2] `https://bugs.eclipse.org/bugs/`, verified 02/01/06
[3] `http://issues.apache.org/bugzilla/`, verified 02/01/06
[4] `https://bugzilla.redhat.com/bugzilla/`, verified 02/01/06

duplicates. The former approach relies solely on expertise; the latter approach also involves an investment of time that might be better spent on other aspects of the project. The manual nature of these approaches also tends to result in missed identifications of possible duplicates due to faulty memory or inadequate searching. These false negatives have several undesirable ramifications: more reports must be managed through the repository for a longer time increasing work loads, developers may miss relevant information in an undetected duplicate report as they are working on the problem, and priorities of reports may have been different had it been clear that a number of people had reported a similar problem.

To help the bug triager, we have developed an approach that recommends for a newly submitted report, a set of potential duplicate reports that already exist in the repository. Based on the textual information entered by a reporter of a bug to describe a problem or enhancement, our approach builds a model of previous reports by creating groups of duplicate reports. When a new report arrives, its textual description is compared to this model to determine the most similar reports, if any. Any reports recommended as duplicates of the new report are returned to the triager who can then make a final decision about whether the reports are indeed duplicates. Our approach is based on techniques from topic detection and tracking research that considers the clustering of news stories [18].

We have evaluated the accuracy of this approach analytically against the Firefox, Eclipse, Fedora and Apache projects. Our best results are for the Firefox project, for which we achieve a precision and recall of 29% and 50% respectively when our tool recommends seven potential duplicate reports. To investigate whether these precision and recall rates help a triager, we conducted a user study in which we had 18 participants, equally split into control and treatment groups, use our recommender on a set of 10 bugs from the Firefox repository.

We found that the treatment group was able to more accurately identify duplicate bug reports, using fewer searches and in slightly less time than the control group, providing evidence that even relatively low precision recommenders might help with this bug triaging task.

We begin by providing background information about bug repositories for open source projects and the process by which duplicate detection occurs today

(Chapter 2). We then review other efforts in this area (Section 2.3) before presenting our approach for recommending duplicates to a bug triager (Chapter 3). We present the results of both an analytic and human-based evaluation of the approach (Chapter 4) and discuss outstanding issues with the approach and our evaluation (Chapter 5). We then summarize our findings (Chapter 6).

Chapter 2

Background

To set the stage for a description of our approach, we provide an overview of the format of bug reports and of the current approaches used by bug triagers to perform duplicate detection. We also discuss some related work on bug reports and duplicate detection research.

2.1 Reports and the Bug Repository

The various bug (or issue) repository systems used by open source projects (e.g., Bugzilla[1], JIRA[2] and CollabNet[3]) share many similarities. A report stored in one of these repositories consists of a number of fields: some fields take free-form text as values; others have pre-defined sets of values from which the reporter of a bug report must choose.[4] For example, a report submitted to a Bugzilla repository has free-form summary and description fields, whereas the values for platform and version fields come from a set of pre-defined values for the project. An example of a bug report is shown in Figure 2.1

In addition to storing information about a specific problem or enhancement, most repositories also support the specification of relationships between reports, such as when one report is a duplicate of another and when one report is dependent upon another. When using a Bugzilla repository, a triager can mark one report as a duplicate of another by changing the status field of the report to `duplicate` and indicating the id of the earlier report. These actions form a symmetrical relationship indicating the duplication between both reports in the repository [14]. This duplicate marking approach can lead to chains of duplicates within the repository. Consider a report B that is marked as a duplicate

[1] `http://www.bugzilla.org`, verified 02/28/06

[2] `http://www.atlassian.com/software/jira/`, verified 02/28/06

[3] `http://www.collab.net/`, verified 02/28/06

[4] In this thesis, we use the term report instead of bug to reflect the use of these repositories for both problem reports and feature enhancement requests and discussions.

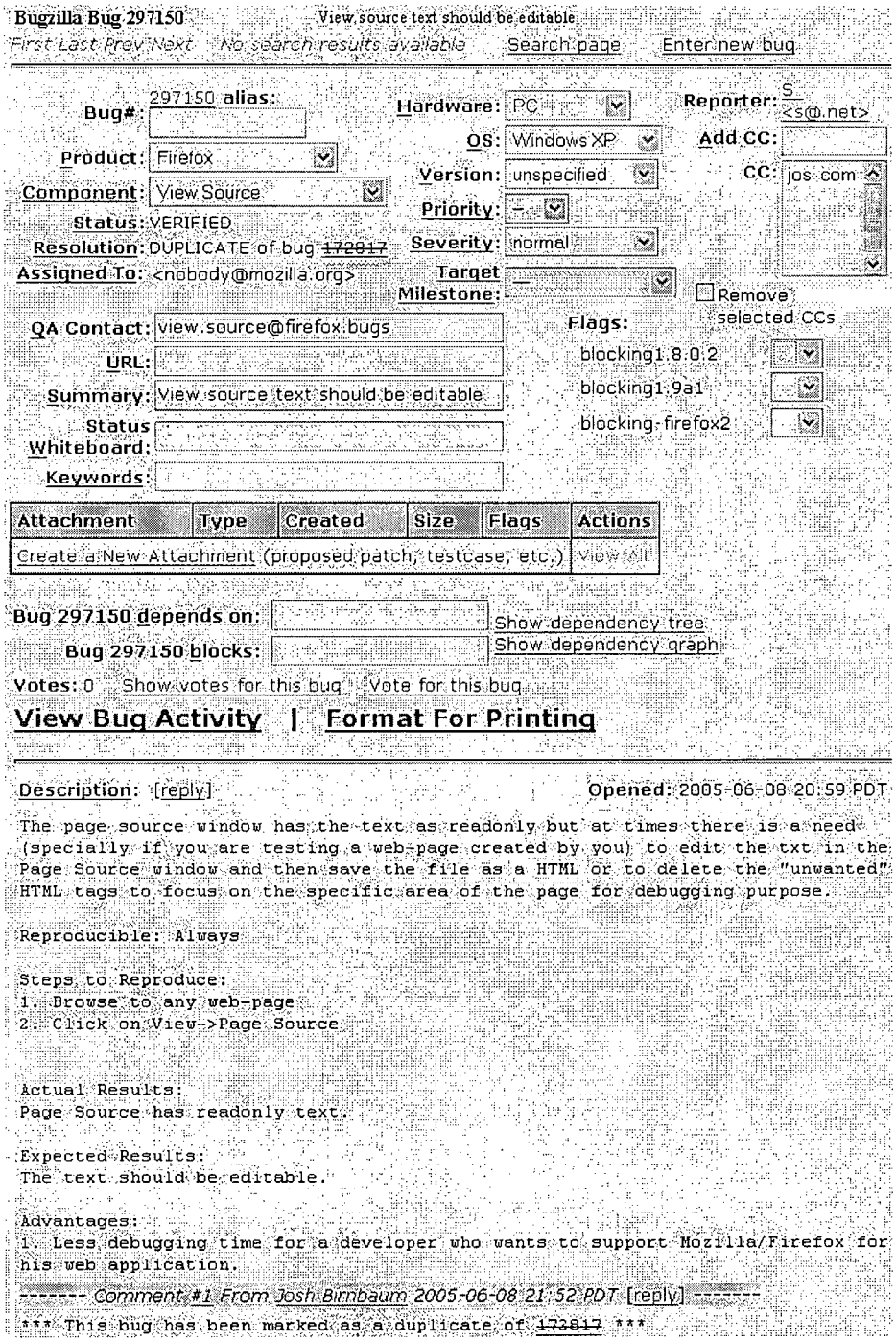

Figure 2.1: Example of a Bug Report from Firefox

of an earlier report A. Later, report C enters the repository and is subsequently marked as a duplicate of report B. By transitivity, report C is also a duplicate of report A, although this is not recorded directly in report A or report C. The presence of such duplicate chains in repositories affects how we apply and validate our approach in Section 3.4 and Section 4.1 respectively.

In this thesis, we focus on reports stored in Bugzilla repositories. Because of the similarities of the repositories used for open source projects, this choice does not restrict the generality of our approach.

2.2 Duplicate Detection Today

In practice, there are two processes currently in use to prevent duplicates in bug repositories.

The first process involves preventing the reporting of duplicates. Many projects ask reporters submitting a report to the bug repository to check if the report being formed is a duplicate before submitting it. For instance, in the Firefox project, the first step instructs reporters to review a posted list of the most frequently occurring reports, and to search for whether a similar report has been submitted in the last few weeks. This process is dependent upon the reporter's motivation to perform extra steps during report submission. It is difficult to assess for a project how many duplicates this prevents in the repository. The number of duplicate reports that exist in open source bug repositories (Chapter 1) suggests that this process is not sufficient.

The second process involves identifying duplicates as a report is being triaged. A bug triager typically attempts this identification by perusing the project's most frequently reported bugs list and by performing searches on the reports in the repository. A commonly used search involves looking for words occurring in the report's summary or its description. More advanced triagers use component names and specialized keywords to aid their searches.

Each of these processes leaves the determination of what is a duplicate up to the human reporter or triager. As a result, there is a wide range of what is considered a duplicate report in open source bug repositories. Some duplicates are obvious, such as a report that has been submitted multiple times. Many duplicates are more subtle. As an example of subtle duplicate reports, consider the Firefox reports #297150 and #269326 that describe feature enhancement

requests.

> *297150*: The page source window has the text as readonly but at times there is a need (specially if you are testing a web-page created by you) to edit the txt in the Page Source window and then save the file as a HTML or to delete the "unwanted" HTML tags to focus on the specific area of the page for debugging purpose.
>
> *269326*: Is it possible to have something like "Edit Source" feature in addition to "View Source" feature. "Edit Source" feature can allow users to change anything in web page source which can be reparsed by Firefox layout engine and re-displayed.

The first report (#297150) asks for the ability to edit text in the page source viewer, while the second report (#269326) asks for this functionality as a separate feature. The first report is marked in the Firefox repository as a duplicate of an earlier report #172817. A bug triager with the Firefox project suggested that the second report, #269326, was also a duplicate of report #172817, but in a conversation recorded on the report, the reporter disagreed, so it was left as a unique bug report. The decision to mark a report as a duplicate or not is not always clearly defined. Duplicate reports also do not always occur within a short time period. An analysis of all Firefox bug reports from project inception until October 2005 shows that the average time between the reporting of the original bug and its last duplicate is 177 days.

2.3 Related Efforts

Perhaps because of a lack of availability to bug repository data until the growth of open source projects, there have been few efforts that have focused on identifying relationships between bug reports automatically. Fischer, Pinzger and Gall [7] introduced an approach that discovers and visualizes dependencies between features in a software system using information stored in bug reports and information about revisions to the code base. Čubranić and Murphy's Hipikat project, which automatically builds a group memory from artifacts recorded during a software development project, determines which reports in a repository are similar to each other based on an information retrieval algorithm [17].

Sandusky and Gasser [14] present an approach that uses duplicate and dependency relationships between reports along with informal references in a report to other reports to extract Bug Report Networks (BRNs), to identify how large, distributed open source projects are managed. Anvik, Hiew and Murphy [2] describe an approach that suggests which developer may be suitable for solving a newly submitted report based on what it learns from assignments made to previous reports in the repository. Compared to these previous efforts, the problem addressed in this thesis is most similar to the bug similarity addressed in the Hipikat project. Duplicate detection differs from bug similarity in requiring a more precise definition of similarity. More specifically, our approach differs from that of Hipikat in that we return the earliest occurring duplicate of a similar bug, if one exists, instead of the bug itself.

Automatic determination of duplicate documents has been considered in other contexts. In large collections of documents, for instance, duplicates are removed to maintain the speed and effectiveness of search engines [4]. In this context, the corpus of documents is often relatively stable compared to a bug repository in which new bugs are constantly being added.

As another example, Topic Detection and Tracking (TDT) research considers the problem of grouping news stories about the same event together [5]. In one formulation of this problem, online event detection [18], the task is to find the first occurring story in a stream of stories seen that describes a new news event; all subsequent stories about the same event are labelled old and ignored. For example, these can be used on a news feed to highlight a new disaster event when it occurs. Online event detection is very similar to our problem of duplicate bug report detection. The approach we take builds on the TDT approach of Yang, Pierce, and Carbonell [18], who use an incremental clustering approach to process each news story as it is received. The main difference with our approach is that we assign a new bug report to its correct centroid or group of duplicates, instead of adding it to the most similar centroid found (Section 3.4). We further extended their approach by adding the centroid and the incremental update scheme used in MailCat [16], a tool for sorting incoming email into an appropriate group of existing folders.

Chapter 3

Recommending Duplicates

To help a bug triager identify more duplicate bug reports efficiently, our approach suggests to the triager whether a new report is UNIQUE or a potential DUPLICATE. In the latter case, our approach produces a set of reports that may be duplicates of the new report.

Our approach relies on building a model of reports in the repository. This model groups similar reports into *centroids*. When a new report arrives, we compare the report to each centroid, looking for occurrences of high similarity; reports from highly similar groups are considered potential duplicates. We build the model incrementally as reports arrive.

Our approach processes each report that arrives at the repository according to four steps:

1. Represent the incoming report as a *document vector* (Section 3.1).
2. Compare the vector to each centroid in the model (Section 3.2).
3. If the similarity between the vector and a centroid exceeds a threshold, mark the incoming report as a DUPLICATE; recommend a report from each of the top n ranked centroids (Section 3.3).
4. Optionally add the incoming bug report to a centroid (Section 3.4).

3.1 Bug Report Representation

We represent a report using the textual values of the report's summary and description fields because these fields contain the most reliable description a new report. The summary field corresponds to a short one-sentence description of the bug. The description field is a longer description that typically includes free-form text describing the problem or enhancement, and for problems, also

includes the steps to reproduce it, the expected and actual behavior, and any additional information.

We combine the values of these fields and preprocess the resulting text, stemming each word with the Porter stemmer algorithm [11] and removing stop words using the Cornell SMART stop list [3]. All words from the description field template in the submission form, such as *Steps to Reproduce:*, *Actual Results:* and the *Build Identifier* string, are also excluded. Finally, we delete any leading and trailing non-alphanumeric characters from all words.

This preprocessed text is then converted into a document vector that is used to represent the report. The vector consists of weighted values w_{ij} for each term j in the document (report) i. We represent the vector as $d_i = (w_{i1}, w_{i2}, ..., w_{in})$, where n is the number of different terms appearing in the current model. For the term weights, w_{ij}, we use Term Frequency - Inverse Document Frequency [12], $(TF - IDF)_j$, where

$$(TF - IDF)_j = tf_j * idf_j \tag{3.1}$$

This weight attempts to capture two characteristics in one value: the term frequency tf_j, which is the number of occurrences of the term j in the document and which favors highly occurring terms that are more likely to represent the context of the document, and the inverse document frequency idf_j, which favours rare terms. The inverse document frequency is computed from the document collection size and the number of documents containing term j.

$$idf_j = log(\frac{\#\ docs\ in\ collection}{\#\ docs\ containing\ term\ j}) \tag{3.2}$$

3.2 Comparing Bug Reports

Our approach maintains a model of previous reports processed and their groupings to allow for comparisons with incoming bug reports. The reports placed within the same group are treated as one aggregate, called a centroid [16]. The centroids in the model we form correspond either to groups of duplicate reports or individual unique reports.

We represent each centroid with a vector, which requires us to compute the weights $(TF - IDF)_{jC}$ for the terms j that appear in the centroid C. This

corresponds to computing the weights for any term j that appears in any document d in the centroid C. The tf_{jC} for term j in the centroid C is computed by summing the term's frequency in each document d within the centroid C:

$$tf_{jC} = \sum_{d \in C} tf_j \tag{3.3}$$

Similarly, idf_{jC} is computed over centroids instead of over documents, using the total count of centroids and the centroid frequency, the number of centroids containing the term j.

$$idf_{jC} = log(\frac{\#\ centroids\ in\ model}{\#\ centroids\ containing\ term\ j}) \tag{3.4}$$

To compare a vector representing an incoming report with the centroids in the model, we use cosine similarity, which is a measure of how similar two document vectors are to each other [12]. For two vectors a and b, the cosine is computed as

$$cos = a \cdot b = \sum_{i=1}^{n} a_i * b_i \tag{3.5}$$

which is the cosine of the angle between the two vectors.

A vector for a report with a long description and a vector representing a centroid with a large number of reports will have larger $TF - IDF$ and cosine values than other centroids. As these values can skew the cosine similarity to favour highly populated centroids (those with many duplicate reports), we normalize the cosine values to fall within the range [0,1]. This is achieved by dividing by the product of the vector lengths of a and b.

$$cos_{norm} = \frac{a \cdot b}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}} \tag{3.6}$$

With normalized values, we can fairly compare vectors for documents and centroids. A similarity of 0 means two vectors have no words in common. A similarity of 1.0 indicates the two vectors have identical proportions of the same set of words and are very likely to represent two exact copies of the same report.

3.3 Labelling Reports Unique or Duplicate

Our approach compares the document vector representing a new report to every centroid that is currently in the model, computing the similarity between each

pair. We rank each centroid according to its computed similarity. We then label the bug report as UNIQUE or DUPLICATE based on whether the top ranked centroid is below or above a predefined threshold. If the similarity of the top ranked centroid is below the threshold, the report is UNIQUE. Otherwise, if the similarity is above the threshold, the bug report is labelled a DUPLICATE.

When the similarity between the incoming report and the top ranked centroid exceeds the threshold, we label it a DUPLICATE and return a list of potential duplicate reports to the bug triager. Because of the large variation in what constitutes a true duplicate report, the list of potential duplicates we return is a representation of the duplicates in the top n ranked centroids. For our evaluation in Section 4.2, we used the values $n = 3$ and $n = 7$. When a top n ranked centroid consists of more than one report, we present the information from the earliest submitted report within the centroid as representative of the centroid. We use the earliest submitted report as representative because the convention in many projects is to mark duplicates with the bug id of the earliest report. Table 3.1 shows a sample of the potential duplicates returned by our approach for the report #297150 from the Firefox project, the report that we introduced in Section 2.2.

Table 3.1: Top 7 Suggested Duplicates for Firefox Bug # 297150

Bug #	**Summary**
221668	View Source is brokens
291603	View source doesn't show the correct source
245851	View Selection Source automatically converts upercase HTML tags to lower case
230693	Missing reload within view source window
279614	Should be able to open view source window in a new tab
172817	Allow external source viewer/editor
285560	a view-source windows should show the source of a streamed page

3.4 Incremental Updates

New reports are constantly being submitted to a bug repository. To fit into this context, our approach must be able to add new reports to the model of

duplicates incrementally. To achieve this goal, we have adapted the incremental approach used in MailCat [16], a system which uses an incremental classifier to predict into which existing folder a received email should be placed.

When a new report is received, and is determined by a triager (hopefully with the help of our approach) to be a duplicate, it is added to the appropriate centroid. Determining which centroid the report should be added to is complicated by several issues. Duplicate reports are not always marked by triagers with the bug id of the earliest duplicate. The report could be marked as a duplicate of a report occurring after it or the report could be part of a duplicate chain (Section 2.1). When adding a report to the model, we can not simply rely on the report's bug id or the marked duplicate bug id. Instead, for each report within a centroid, we record its bug id and the duplicate bug id it is marked with. Before a report is added to a new centroid, we first check if one already exists for it. This is done by comparing the bug id of the report, and if a duplicate, the bug id of the duplicate, to the recorded bug ids in each centroid. If a match is found with an existing centroid, the report is added to that centroid. If no match is found, a new centroid is created for the report. This ensures that a report is added to its correct centroid, instead of being added to a new one.

The $(TF - IDF)_{jC}$ weights used in our cosine similarity values are computed using the current values for term frequency (tf_{jC}, the frequency of term j in centroid C), centroid frequency (# of centroids containing term j) and the total number of centroids. When a report is added to either an existing or new centroid, these values must be updated. Term frequency is updated by adding the term frequencies of the new report to its corresponding centroid. If any new terms were added to the centroid, we update the centroid frequency for each new term j. This is accomplished by incrementing the centroid frequency by 1 for each new term j. Similarly, if a new centroid was created for the report, the total number of centroids is incremented by 1.

Updates in the bug repository can also be quickly updated in the model. If a bug report is deleted or marked as being `INVALID`, all that needs to be done is to subtract its term frequencies from the centroid, decrement the centroid frequencies for words no longer appearing in the centroid, update the total number of centroids, and delete the report's bug id and duplicate bug id from its centroid. Changes to a bug report's summary and description can also be updated in a similar fashion.

3.5 Performance

We designed our duplicate detection system to be incremental, to handle the dynamic nature of large, open source bug repositories. Reports are constantly added to and updated in the repository. If desired, our detection approach can be tuned to update the model of previous reports in batches, for example, after every 50 reports, or every day at 2AM. For a bug repository, the model can also be populated with the existing reports, all in one large batch. This method was used for our user study (Section 4.4) to build a model of all Firefox bug reports submitted before June 9, 2005.

An important factor to the usability of an approach is its speed. Updating the model with a new report is quick with our incremental approach, requiring less than a second of time. When the model was built for our user study (Section 4.4), using approximately 18,000 reports, the process took about 30 minutes, which is about a 1/10 of a second for each report. This is an acceptable speed, considering that we also preprocess the report's text before adding it to the model. The amount of time needed to determine the recommendations is also important. Although this largely depends on the number of reports in the model, less than 2 seconds are needed to provide the recommendations for a bug report in the user study.

3.6 Implementation

We have implemented our approach as a Java application. Figure 3.1 shows a screen shot of the system used in the user study (Section 4.4). It displays the recommendations for the current bug report being viewed. The triager can view both the report and the recommendations without the need to switch back and forth between the two. The triager can also double-click on a bug report from the recommendations to open it in the current web browser window. We discuss the implementation of the duplicate detection system along with details on how the backend of the system was implemented.

Initially, the bug reports are stored in XML files, chronologically ordered, and are read into the system in the same order. As each bug report is read into the system, we retrieve its bug id. We then scan the comments for any indication that the report has been marked as a duplicate. In Bugzilla, when a

Figure 3.1: Screenshot of Duplicate Bug Report Recommender.

report is marked as a duplicate, a message is added with the following text: *** *This bug has been marked as a duplicate of bug <bug_id>* ***, where <bug_id> is the id of the duplicate report. When one such comment is found, we extract the bug id of the duplicate using regular expressions.

We begin preprocessing the new bug report's text by removing any strings in the bug report description that are added from Bugzilla's description field template. For example, if a bug reporter left the description empty, the report's description would appear as the following:

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.2
Build Identifier: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.8.0.2) Gecko/20060308 Firefox/1.5.0.2

Reproducible:

Steps to Reproduce:
1.
2.
3.

Actual Results:

Expected Results:

Any strings from the above template that are found in the bug report description are removed. The `User-Agent` and `Build Identifier` strings are matched using regular expressions.

We concatenate the bug report summary with its description text to form a string. All characters are converted to lowercase. Then the string is tokenized, using any consecutive number of whitespaces as the delimiter. The resulting tokens become the terms we process. To remove common stop words, we retrieve the words contained in the SMART system's stop list [3]. As each stop word is read from the stop list file, it is stored in a hash, for quick access. As we view each term in the string, any term that exists in our hash of stop words is removed. The remaining terms are stemmed, to change them into their root form. Each term is stemmed using the Porter stemming algorithm [11]. The processed terms are then used to build a document vector representation of the bug report.

3.6.1 hash Tables as Document and Centroid Vectors

The processed bug report text, which includes the summary and the description, is stored as a hash table. The hash table represents the document vector for the

report. In the hash table, each key and value entry represents a term and its the number of occurrences of the term in the report. While adding each term to the hash table, if an entry already exists for a term, we increment its existing value by one. The hash table is used to represent the term frequencies for each term in the bug report.

For the centroids, we maintain a count of the current number of centroids. Each centroid is also represented by a hash table. Each hash table entry consists of the term as the key and its frequency in the centroid as the value. In addition, the centroid frequencies and the number of centroids containing some term j, are also stored in a hash table. The term j is used as the key and the value is the centroid frequency. The use of the hash table allows us to quickly retrieve the term and centroid frequencies in order to compute the $TF - IDF_{jC}$ weights for each term j. If the number of centroids is zero, we return a cosine of 0. To allow for quick retrieval of a centroid's hash table, a reference to it is stored in a hash table, using the centroid's first seen bug id as the key (explained in more detail below).

To keep track of which bug ids have been seen so far, we store them in lists, one for each centroid. Both the bug ids and the duplicate bug ids are stored. When a new report is encountered, we retrieve its bug id and its duplicate id, if one exists. Then, we scan each centroid's list of bug ids. If a match is found between the report's bug or duplicate id and any id from the list, the new report belongs to that centroid. Both the bug id and the duplicate bug id are then added to the end of the centroid's list. To ensure the new report is added to the correct centroid's hash table, its key, the first occurring bug id in its list of ids is retrieved. On the other hand, if no matches are found for the new report's bug or duplicate id, we create a new list for the new centroid and add the report's bug id and duplicate id to the list.

Finally, we need to compute cos_{norm}, the normalized cosine value. This requires computing the lengths of both the report's and the centroid's vector. We then divide the cosine by the product of these vector lengths. If the product is 0, we return a normalized cosine value of 0.

3.6.2 Updating the Model

Updating the model's hash tables is easily done. Retrieve the hash table value for the given term, update the value and store it back in the hash table. This

allows for efficient updating of term and centroid frequencies. When a new report is added to the model, the hash table representing the term frequencies of the bug report is used to update the centroid frequencies in the model. When determining whether the centroid frequency for a term j should be incremented, we check if the term already exists in the centroid's hash table. If it does not, we increment the centroid frequency for that term, because another centroid now contains the term j.

3.7 Performance Optimization

There are a few optimizations that can be made to increase the speed of the recommender. The method we have used has a time complexity of $O(n^2)$. As the number of reports contained in the model increases, the time required to find the recommendations increases quadratically. With our Firefox dataset containing over 21,000 bug reports, about 10 hours on a 1 GHz machine was needed to simulate the approach on the whole dataset. We present the optimizations we used and their performance.

3.7.1 Sparse Vector Computation

For our approach we use the full vocabulary of all terms seen so far by our model. Even after a few thousand reports, the vocabulary can quickly grow to over 10,000 terms. With each term corresponding to a dimension in the vector, computing the cosine with such large vectors is time consuming. Fortunately, many terms do not occur in every report or centroid, making the $TF - IDF$ weights for these terms be 0. This creates a sparse vector for our reports and centroids, where only a few values are non-zero. Because of this, we can take advantage of the sparseness and compute the cosine using a faster method [15].

With sparse vectors, only non-zero dimensions make a contribution to the resulting cosine measure. The cosine measure is essentially the dot product of the two vectors. Given two vectors a and b of equal dimensions, this is computed by summing all values of $a_i * b_i$, for each dimension i. For any dimension i where a_i or b_i are 0, $a_i * b_i$ will also be 0. For sparse vectors, we can reduce some of the computation by only computing $a_i * b_i$ for dimensions i where both a_i and b_i are non-zero. This is implemented by scanning all the terms in the bug report's

hash table and checking if it also exists in the centroid's hash table, before computing the dot product.

Another performance optimization that can be done is when normalizing the cosine value. The vector length of both the centroid and document vector must be determined to normalize the dot product value. For a vector a, the length is obtained by summing a_i^2, for each dimension i. Similarly, we can avoid many of the 0 values, by computing a_i^2 for only the non-zero values. Since both the centroid and bug report hash tables only contain values for occurring terms, we only need to compute a_i^2 for terms occurring in the hash tables.

3.7.2 Using the Top 20 Terms

To speed up the time needed to compute the recommendations, especially when the model contains 10,000 or more centroids, we only make comparisons with a subset of centroids. Given a new report, we extracted the top 20 terms, ranked by idf_{jC} (Section 3.2). We have used 20 terms because this was determined to be an optimal number of terms to use [8]. We then fetch all centroids containing at least one of the 20 terms, and compute the recommendations based on this subset of centroids. This reduced the total computation time by approximately 30% for each project.

Chapter 4

Analytic Evaluation

As an objective evaluation of how well our approach can determine duplicate reports, we applied it to data in the bug repositories of four open source projects: Firefox, the Eclipse Platform, Apache 2.0 and Fedora Core.

We chose these four projects because they are mature and popular, cover a diverse set of user communities, are implemented in different programming languages and are built using different organizational processes.

For each of these projects, our evaluation consisted of replaying the submission of reports to these projects. For each report, we applied our approach to recommend if the report was UNIQUE or a DUPLICATE, evaluated our recommendation against the report's true identity, and then updated the model used in our approach with the actual report. For our evaluation, a report's true identity consists of its correct label of either UNIQUE or DUPLICATE, and for a DUPLICATE, includes the bug id of a previous duplicate report. This method of evaluation simulates how our duplicate detection approach would be used within a triaging environment. In this section, we detail our evaluation method (Section 4.1), and assess the performance of our approach using standard information retrieval measures (Section 4.2) as well as measures from the Topic Detection and Tracking (TDT) community (Section 4.3). Then we discuss the user study of our approach (Section 4.4) and the threats to its validity (Section 4.5)

4.1 Method

For each project, we used a selection of bug reports from the beginning of the project until Sep. to Oct. 2005 (Table 4.1 lists the exact end dates used for each project). We use a selection of reports because not all reports may proceed to be triaged for duplicates. For instance, bug reports with resolutions of WORKSFORME, indicating a reported bug cannot be reproduced, INVALID, indi-

cating the reported bug is not a bug, and `WONTFIX`, indicating the report will not be attended to, may preempt any detection of duplication. The reports we used are ones with resolutions of `FIXED`, `DUPLICATE` or `OPEN`[1]. The Fedora Core project includes additional resolutions for fixed bugs which we also included, namely `RAWHIDE`, `CURRENTRELEASE` and `NEXTRELEASE`; we considered all of these resolutions as equivalent to `FIXED`.

Table 4.1: Time Span of Bug Report Sets

	End Date of Report Set
Firefox	Sep 1, 2005
Eclipse Platform	Sep 29, 2005
Apache 2.0	Oct 14, 2005
Fedora Core	Sep 29, 2005

Since bug report ids increase incrementally over time in the project repositories, we process the reports for each project in order of their ids. For each report (in order), we perform the following three steps:

1. Apply our approach to the report, labelling whether the report is UNIQUE or DUPLICATE against a model formed from the reports processed to that point.

2. Compare the label. If it is a DUPLICATE, also compare our recommendations to its actual identity to determine if they were appropriate.

3. Use the report's true identity (not our assigned label) to update the model of reports used in applying our approach.

The decision to use the report's actual identity to update the model, rather than the recommended label, is based on how bug triage is done. In the four projects we used for evaluation, we looked at how many comments a bug report received before being marked as a duplicate. In each of the projects, approximately 90% of duplicates were marked as `DUPLICATE` before any comments were added. This data indicates that most duplicate reports are identified during the first triage attempt by a triager. Since our approach is semi-automated, the triager will make the final decision on a report's identity. We choose to rely on

[1] we use the label `OPEN` for clarity; the actual value is a blank field

the triager's decision because this information is available to the system during the triaging process and ensures the model of past reports stays consistent with the bug repository.

On the surface, it appears simple to determine if the label, and for duplicates, the recommended bug ids, we assign to a report using our approach are correct; simply check in the bug repository if the bug was recorded as a `DUPLICATE`. Unfortunately, determining the true identity of a report is not this simple because of three situations that occur. First, a duplicate is sometimes marked with the id of a future report, one that occurs after it in time. In this case, in the chronological processing of reports, the current report effectively becomes unique, because the future bug report has not yet been seen at that moment in time. Also, the future report becomes a duplicate, because its bug id has been referred to by a past report. Second, a duplicate can be marked with the id of a report that does not appear in the range of data used in the evaluation. Third, the previous duplicate bug id referred to by the report can be changed several times, in which case we choose the last marked bug id as the correct bug id of its duplicate.

To allow for a fair evaluation of our approach, we must handle these cases in the evaluation. We thus base our determination of whether we consider a bug report we are processing as an *actual duplicate* on the following steps. As we process each report, we record its id. If the report is marked as a `DUPLICATE` in the repository, we also consider the id of which it is marked to be a duplicate, which we refer to as the *duplicate-of* id. Both of these ids are recorded because future reports may refer to either of these ids[2]. For reports with a `DUPLICATE` resolution, if the duplicate-of id is outside the range of data used in the evaluation, we consider the report as unique. Otherwise we check if either the bug id or duplicate-of id has been seen in our processing. If either produces a match, we consider the report a duplicate, for the purposes of evaluation. If no matches are found, the `DUPLICATE` report is considered unique, as the system would not be able to correctly detect it as a duplicate. In our evaluation, for a report we label as a `DUPLICATE` to be a *correct duplicate*, the report must be an actual duplicate and either its bug id or its duplicate-of id in the repository must match a bug id listed in one of the centroids returned within the top n recommendations.

[2]See Section 2.2 for an example of 2 bugs both referring to a third.

The initial number of unique and duplicate reports is listed in Table 4.2. The adjustments reduce the actual number of duplicates that can be correctly predicted by a system, and is shown under the Predictable Dupes column in Table 4.3. These values list the number of reports and its percent of the total reports.

Table 4.2: Initial Bug Report Breakdown

	Unique	Dupe		Total
		#	%	
Firefox	11096	10819	49	21915
Eclipse Platform	29063	8653	23	37716
Apache 2.0	1370	412	23	1782
Fedora Core	17961	4115	19	22076

Table 4.3: Adjusted Bug Report Breakdown

	Unique	Predictable Dupes		Total
		#	%	
Firefox	13845	8070	37	21915
Eclipse Platform	30354	7362	20	37716
Apache 2.0	1432	350	20	1782
Fedora Core	18846	3230	15	22076

4.2 Results: Information Retrieval Measures

One way to assess our approach is to use measures common in information retrieval: precision, recall and the F_1 measure [15]. These are calculated with respect to duplicate reports. We used the values of $n = 3$, following MailCat [16], and an arbitrary value of $n = 7$ in our study. In this context, we define precision as the fraction of correct duplicates (number of correct answers) out of the number of reports recommended as DUPLICATE by our approach (Equation 4.1). We define recall as the fraction of correct duplicates (number of correct answers) out of the number of predictable duplicates (Equation 4.2). Since the labelling of a report in our approach depends upon the threshold value for similarity, it

can be varied to obtain higher recall with lower precision values or vice versa. The F_1 measure is used to balance the inevitable tradeoff between precision and recall (Equation 4.3).

$$Precision = \frac{\#\ correct\ duplicates}{\#\ reports\ recommended\ as\ duplicate} \tag{4.1}$$

$$Recall = \frac{\#\ correct\ duplicates}{\#\ predictable\ duplicate\ reports} \tag{4.2}$$

$$F_1 = \frac{2 * precision * recall}{precision + recall} \tag{4.3}$$

The precision and recall values for the highest F_1 scores, obtained by varying the threshold value, are show in Table 4.4. The precision (%) and recall(%) for the top 3 and top 7 recommendations are shown, along with the threshold values. Compared to the state-of-the-art precision and recall values in Information Retrieval (IR), our values are low, which is likely due to the difficulty of the duplicate detection task (Section 5.1).

Table 4.4: Accuracy on Duplicate Bug Reports

	Top 3			Top 7		
	prec.	rec.	thres.	prec.	rec.	thres.
Firefox	27	36	0.4	29	50	0.35
Eclipse	12	17	0.5	14	20	0.5
Apache	23	30	0.35	24	32	0.35
Fedora	24	21	0.6	19	31	0.5

4.3 Results: Detector Accuracy Measures

There are many other measures that can be used to evaluate our approach. We wish to not only correctly recommend duplicate bug reports, but to also predict whether a report is unique. We can use the duplicate and unique detection accuracies to see how well the system handles both of these requirements. Duplicate accuracy is the number of correctly detected duplicates out of all predictable duplicates (Equation 4.4). Likewise, unique accuracy is defined as the number of correctly labelled unique reports out of the total number of unique reports

(Equation 4.5). Duplicate accuracy is equivalent to the recall value used in the previous section (Section 4.2) because we prefer to have more duplicate reports being correctly detected, rather than having high precision.

$$Duplicate\ Accuracy = \frac{\#\ correct\ duplicates}{\#\ predictable\ duplicates} \tag{4.4}$$

$$Unique\ Accuracy = \frac{\#\ correct\ unique\ reports}{\#\ unique\ reports} \tag{4.5}$$

By varying the threshold value used, different duplicate and unique accuracies can be obtained. Figure 4.1 shows this tradeoff for the Firefox project. The other 3 projects are displayed in Figures 4.2, 4.3 and 4.4. unique represents the unique accuracy and top 7 and top 3 represent the duplicate accuracy for the top 7 and top 3 recommendations. We can vary the threshold value to obtain a high accuracy duplicate or unique detection system to suit the application and the preference of the triagers. If a high duplicate accuracy is desired, the threshold can be set to a low value, while sacrificing unique detection performance. Vice versa, if one wishes to have good unique accuracy, the threshold can be set to a higher value, but duplicate accuracy will suffer.

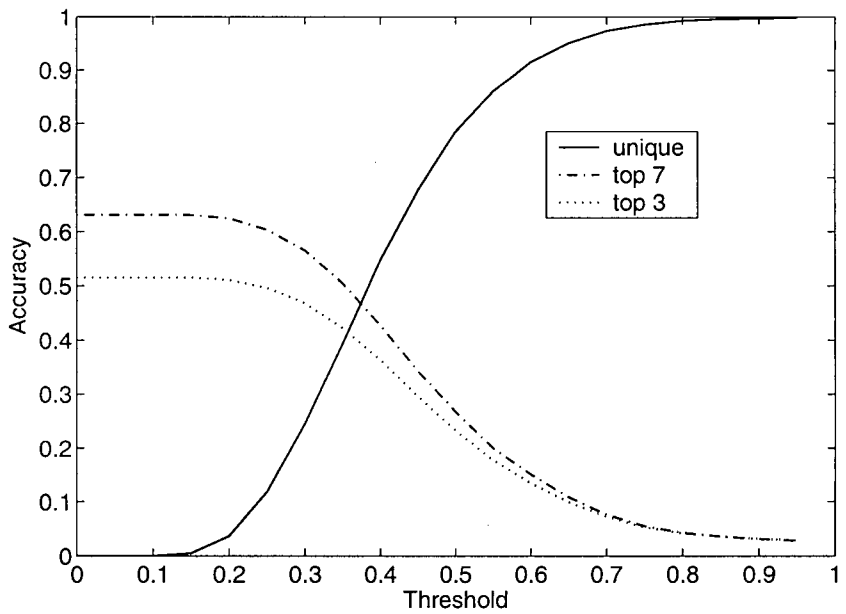

Figure 4.1: Unique/Duplicate Accuracy vs Threshold for Firefox

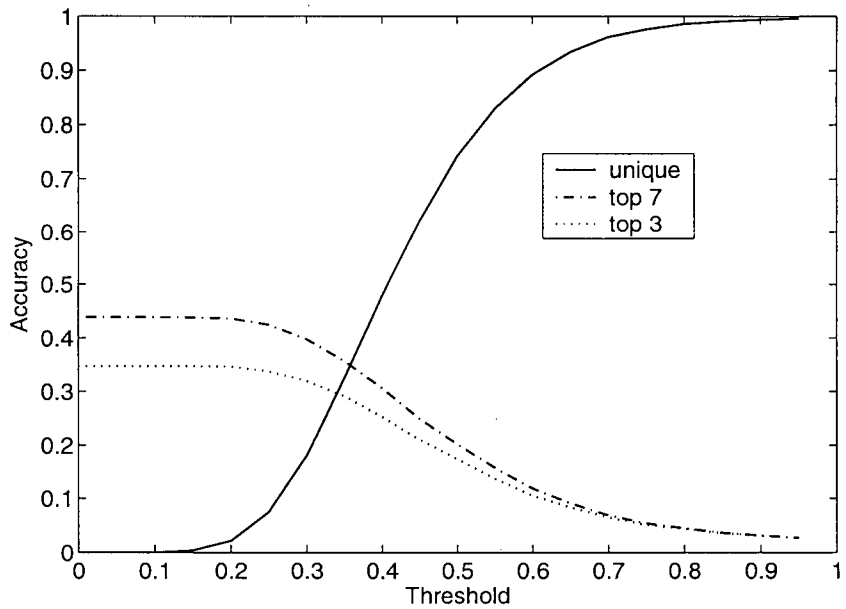

Figure 4.2: Unique/Duplicate Accuracy vs Threshold for Eclipse Platform

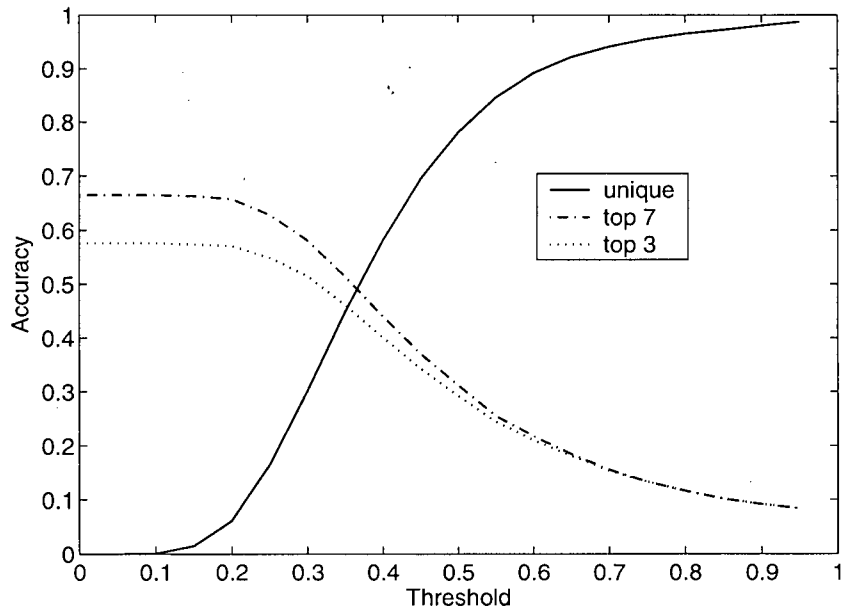

Figure 4.3: Unique/Duplicate Accuracy vs Threshold for Fedora Core

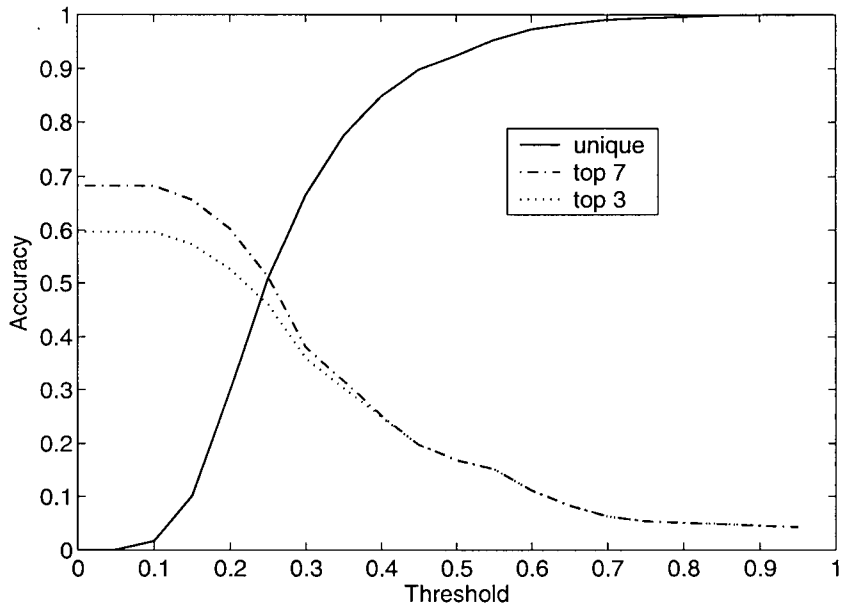

Figure 4.4: Unique/Duplicate Accuracy vs Threshold for Apache 2.0

All 4 projects display a similar tradeoff behavior, although there are some differences between them. Apache has the quickest drop in duplicate accuracy as the threshold value is increased, whereas the other projects show that the tradeoff changes more gradually as the threshold is varied. Apache also achieves the highest possible duplicate accuracy rate out of the 4 projects, with Fedora following behind it.

Another aspect to examine is how the approach performs over time as more bug reports are added to the model. This can be accomplished by looking at the running accuracy of the system. For running accuracy, we simulate the order of submission of the reports, and measure the duplicate accuracy for all the bug reports seen up to that moment in time. The duplicate accuracy is measured after every 50 bug reports. The running accuracy for each of the 4 bug repositories is displayed in Fig 4.5, 4.6, 4.7 and 4.8. The threshold was set to 0.2 in all cases.

Each of the 4 bug repositories exhibit similar behaviors. Initially, they each have a high accuracy for the first few thousand of bug reports. As addition reports are added to the model, the duplicate accuracy decreases. Eventually,

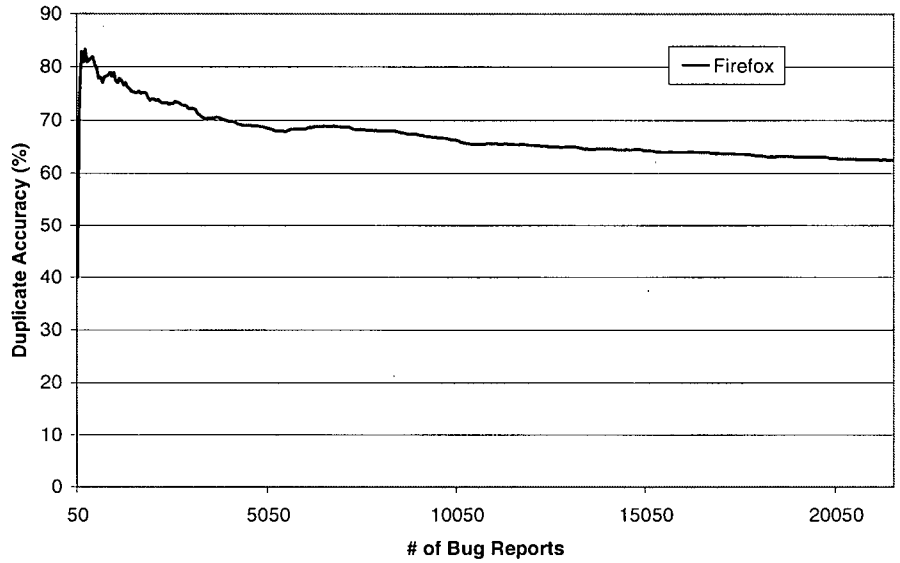

Figure 4.5: Running Accuracy with Top 7 Recommendations for Firefox

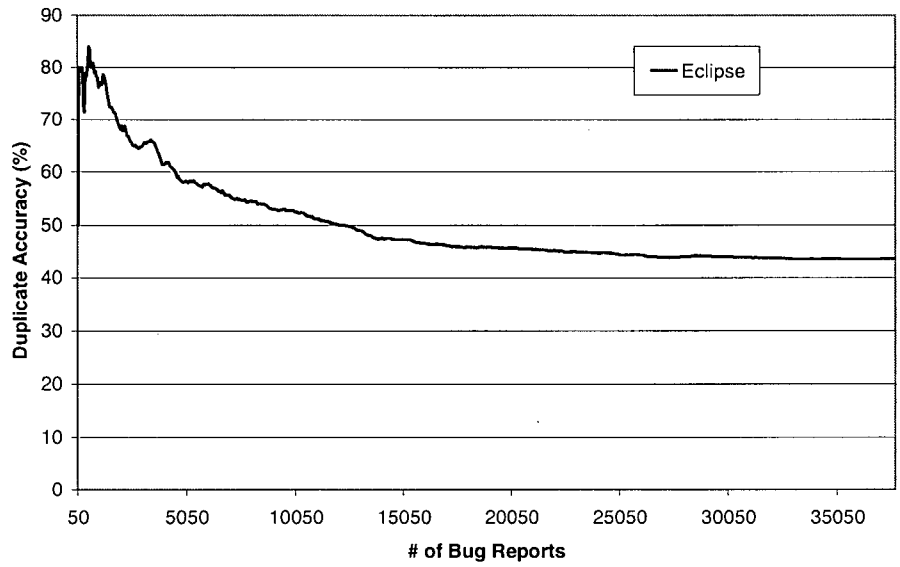

Figure 4.6: Running Accuracy with Top 7 Recommendations for Eclipse Platform

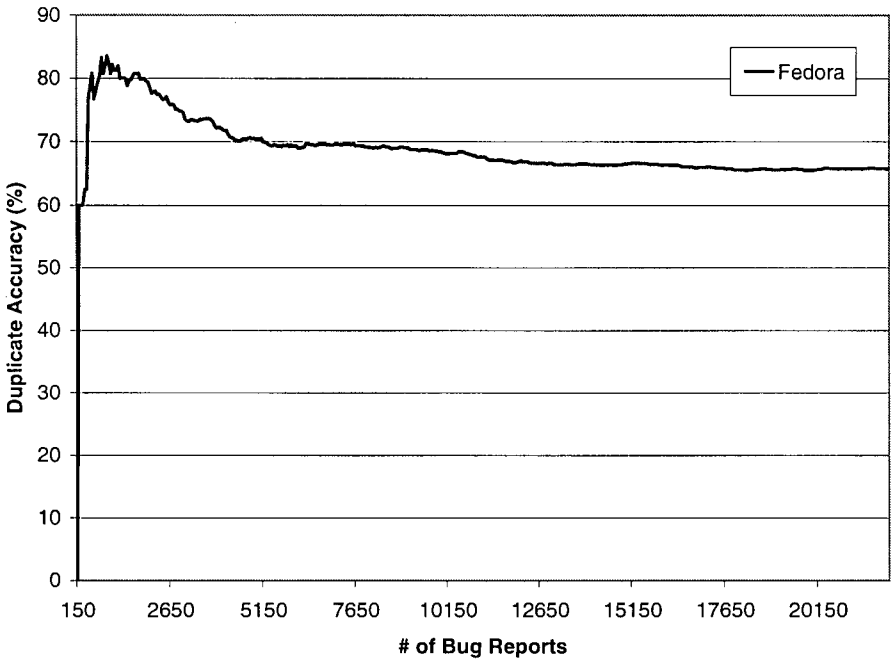

Figure 4.7: Running Accuracy with Top 7 Recommendations for Fedora Core

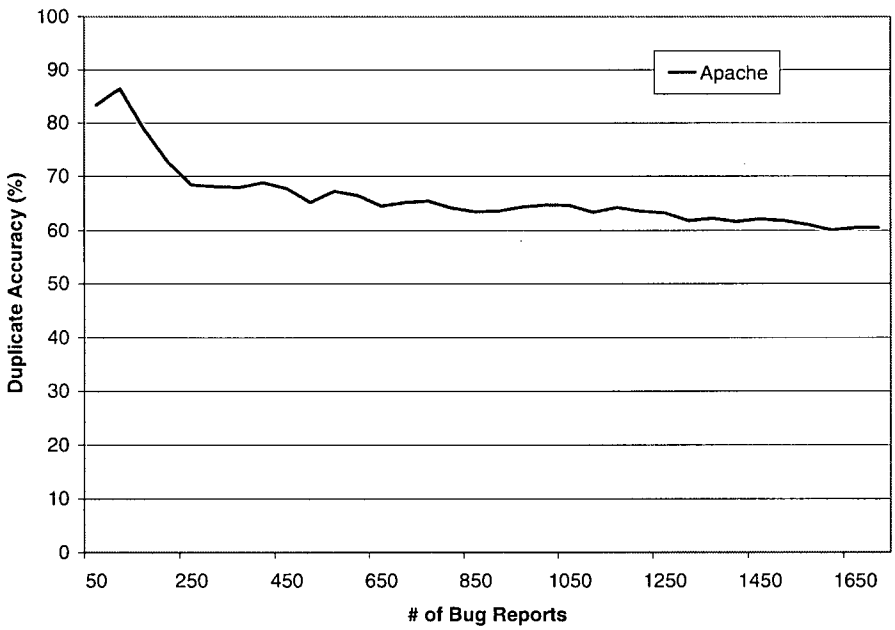

Figure 4.8: Running Accuracy with Top 7 Recommendations for Apache 2.0

the accuracy appears to stabilize and level off. This seems to indicate that the performance of the approach degrades gracefully as the number of reports in the model increases.

Since our approach is similar to that used in event detection research (Section 2.3), we also consider how well our approach performed according to a standard measure in that field, a Detection Error Tradeoff (DET) curve [10]. A DET curve plots the miss versus the false alarm rate as the threshold is varied. In terms of duplicate detection, a miss is a duplicate that was labelled as UNIQUE (Equation 4.6), and a false alarm occurs when a unique report is labelled a DUPLICATE (Equation 4.7).

$$Miss\ Rate = \frac{\#\ duplicates\ labelled\ unique}{\#\ predictable\ duplicates} \tag{4.6}$$

$$False\ Alarm\ Rate = \frac{\#\ unique\ reports\ labelled\ duplicate}{\#\ unique\ reports} \tag{4.7}$$

In the TDT setting, the miss and false alarm rates are computed on the news stories concerning a set of news events. News stories not related to the set of events are ignored and not included in the results. Likewise, our miss and false alarm rates are based only on the groups of duplicate bug reports. The earliest report in a group of duplicates is considered the unique or new event, with the remaining reports being duplicates or old. Figure 4.9 shows the DET curves that result from applying our approach to the four projects. An ideal system would have a zero miss and false alarm rate, so the closer the DET curve is to the origin, the better the detection system.

The DET curves for all the projects are similar, indicating our approach performs similarly, when applied to each project. The curve also shows the tradeoffs that can be made when optimizing our detection system for a particular application. If a bug triager is only concerned about missing duplicates, the system can be tuned to have a low miss rate, while producing many false alarms. On the other hand, if one wants to minimize false alarms, this can be done at the cost of increasing the number of missed duplicates.

4.4 User Study

Ideally, a duplicate detection system would achieve near 100% accuracy and it would be possible to run the detection system as bugs are reported and

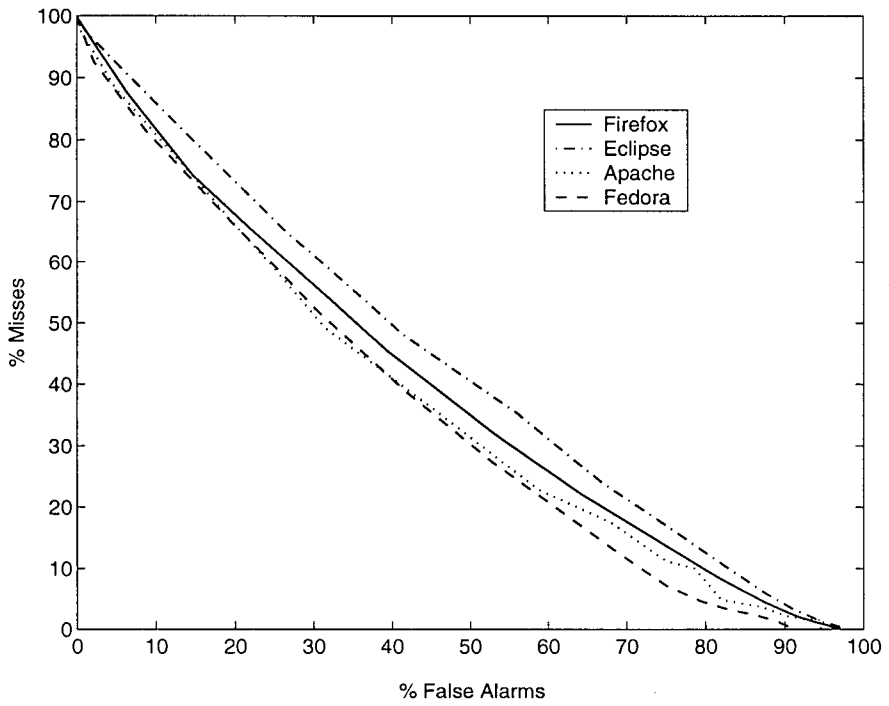

Figure 4.9: DET Curve for Varying Threshold Values

to remove duplicate reports automatically without ever presenting them to a triager. Any lower accuracy in the duplicate detection system means that it is necessary to present the triager a list of possible duplicates from which they might choose. The duplicate detection system thus serves as a recommender. By providing recommendations of duplicates, we would hope to increase the duplicate identification accuracy of bug triagers and possibly reduce both the number of searches performed and the time needed to find duplicates.

4.4.1 Method

Our study consisted of 18 subjects triaging 10 bugs from the archives of the Firefox project. Half of these subjects formed the control group, attempting to detect duplicates through searches on the bug repository. The other half of the subjects formed the treatment group; these subjects were asked to consider recommendations from our duplicate detection system before considering searching

the repository.

The study was conducted remotely. Each subject was randomly assigned to either the treatment or control group. Once assigned, the subject was presented a link to download the correct version of a Java application to use for the study.

Each subject was asked to go through a set of training web pages. Both groups' training consisted of an instructional web page from the Mozilla project on finding duplicates. The treatment group was given an additional tutorial on using the duplicate detection tool. Each subject was then presented four bugs on which to train (Firefox bug reports #297147, #297175, #297196 and #297372). Being instructed to spending no more than 10 minutes per bug, the subject was asked to determine if the bug was a duplicate, was possibly a duplicate, or was unique. If the bug was a duplicate or a possible duplicate, the subject was asked to provide the id of the (potentially) duplicated bug[3]. We allowed a subject to label a report as a possible duplicate because bug triagers are not always sure that the bug report they've found is a duplicate. In this case, a triager will add a comment to the bug report with the bug id of the possible duplicate, asking someone else with more experience for confirmation. Of the four training bugs, three had duplicates in the repository and one was unique. Each subject was provided with the answers for the training bugs and a description of searches that could be used to find the duplicates.

Following the training, each subject was given a pre-test of two duplicate bug reports (Firefox bug reports #297197 and #297380). Our criteria for passing is to correctly label at least one of the two bug reports. 17 out of 18 subjects passed the pre-test. On further examination, the subject that apparently failed the pre-test had actually skipped it. Considering that all 4 training and 10 study bug reports had been completed, we decided to include their results in our analysis, on the assumption that the pre-test had been accidently missed.

Each subject was then presented with a list of 10 Firefox bugs to triage for duplicates. We refer to these reports as #1 to #10. The mapping between an assigned number of 1 to 10 and its bug report number is provided in Table 4.5.[4] The subject was presented the list of bugs in random order to help reduce any

[3]The subject was restricted to listing one bug as being the actual duplicate even though there might be more than one duplicate in the repository. There was no limit on possible duplicates

[4]These bug reports were submitted to the Firefox repository during June 9-10, 2005. Only reports with resolutions of `FIXED`, `DUPLICATE`, or `OPEN` were used.

effects of learning from a previous bug. Nine out of the ten bug reports were duplicates. Subjects were not informed of the ratio of duplicate to unique reports and were only told that "not all bugs are duplicates." To give the recommender a duplicate accuracy close to maximum, as seen in the Firefox repository (see Fig 4.1), we used a threshold of 0.2. The study matched this level of accuracy with five of the nine duplicate bugs having correct recommendations. Correct recommendations occurred at different ranks within the recommendation list; some appeared as the first or second bug while others appeared at the sixth position. As in training, subjects were told to limit themselves to ten minutes of searching per bug report, in an effort to prevent them from spending too much time on difficult reports.

Table 4.5: Mapping of Bug Report Ids

Assigned Bug Report #	Actual Bug Report #
1	297149
2	297150
3	297221
4	297276
5	297306
6	297181
7	297272
8	297307
9	297320
10	297321

All of a subject's actions were recorded as they worked on a bug, including bug reports viewed and bug searches performed. When a subject completed the study, the history of their actions was sent to a server at our university. During the process of sending the history log, subjects were also given an opportunity to answer a short questionnaire about their experience with triaging bugs and with using the tool.

When searching the repository, the subjects used the live Firefox Bugzilla web site from within our Java application. Since the bug reports used in the study are old reports from Firefox, viewing the actual report would reveal the actual duplicate/unique marking of the report. To emulate an environment

where subjects would see only the information a bug triager would see in a newly submitted reports, we filtered all bug reports and searches accessed through the browser. For a bug report, any status and resolution fields were replaced with initial values. Information about patches, dependencies, and any additional comments occurring after June 9, 2005 were removed. The Bugzilla searches were modified so only bug reports created before the June 9 cut off date were returned.

4.4.2 Participants

The ideal subject for our study would be familiar with the technology target of the Firefox project and terms associated with browsers, but would have no triage experience. We chose this profile because we wanted to test our system with subjects who would be most comparable to new bug triagers. Bug triaging is used as an initial activity to get people involved in an open source project, but as they gain experience, they usually move on to other duties, such as testing or programming.

To attempt to recruit ideal subjects, we advertised our study at several Canadian universities, calling for participants who had completed a software engineering course and an object-oriented programming course. This requirement was to ensure that subjects were able to comprehend short code snippets and terminology found in some of the bug reports. We recruited seventeen graduate students and one undergraduate student in this way. Subjects were randomly assigned to either the control or treatment group, with nine subjects in each group.

Each subject was informed that study would take about 1 to 2 hours to complete. A reimbursement of a $20 coupon for a book or music e-commerce site was provided to each subject who completed the study. Five of the 18 subjects indicated in the questionnaire that they had previously triaged bug reports, with three being in the control and two in the treatment.

4.4.3 Results

We had three hypotheses about the use of our duplicate recommender tool: (1) it would lead to higher accuracy when determining duplicates, (2) it would reduce the workload of the triager, and (3) it would reduce the time for the

duplicate detection portion of triage activities. To investigate these hypotheses, we analyzed the logged actions of the subjects when triaging the 10 bug reports presented after the training and pre-test. Of the 18 subjects who uploaded data to our server, we analyzed the data of 9 subjects from the control group and 9 from the treatment group.

Figure 4.10 shows the distribution of scores—correctly marked bug reports—in the treatment and control groups. (Scores are described in more detail below.) For each group, this plot shows that the distribution of scores is non-normal. As a result, we rely on a qualitative analysis of the data.

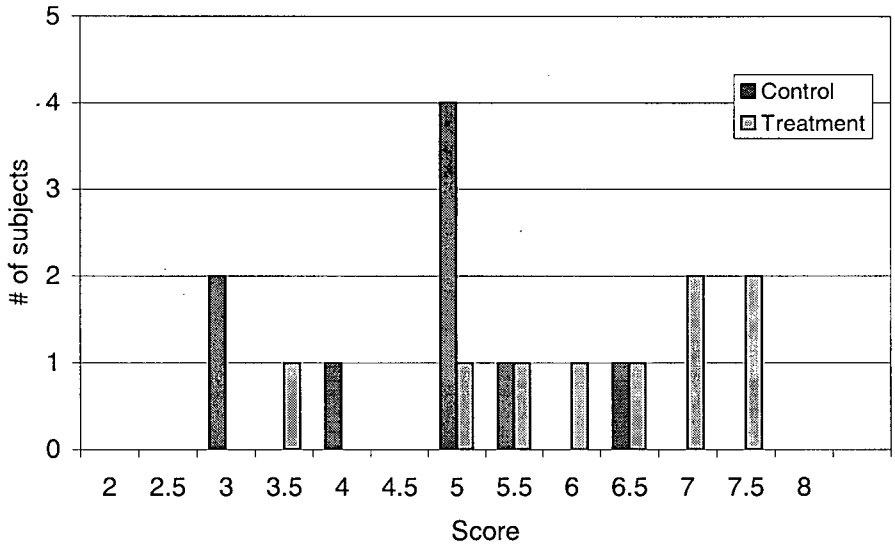

Figure 4.10: Distribution of Subject Scores

4.4.4 Accuracy

The first hypothesis of interest is whether our tool can improve the accuracy of triagers in finding duplicates. We scored each subject's results based on how many reports they triaged correctly. One point was given if the report id specified as a duplicate was one of those marked as such in the repository. For a unique report, a point was awarded only if no reports were marked as duplicates. A duplicate report with an incorrectly marked duplicate, but with one correct possible duplicate was awarded 0.5 points. No penalties were assigned for incorrectly marked duplicate or possible duplicate reports.

Figure 4.11 shows the accuracy for the control and study groups, for each bug report. Each bar represents the group's accuracy (Equations 4.8) for each bug report.

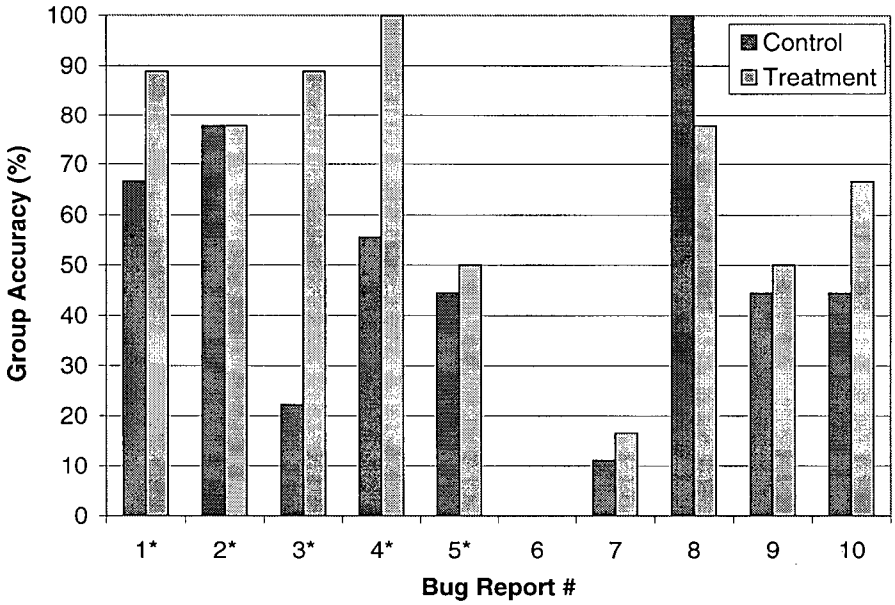

Figure 4.11: Subject Scores by Bug Report

$$Group\ Accuracy = \frac{Group's\ Total\ Score * 100}{Group's\ Max\ Possible\ Score} \tag{4.8}$$

For example, if there are eight subjects in the control group, and five subjects received one point and two subjects received 0.5 points, then the group's total score is six. Thus the group's accuracy is $(6 * 100)/8$ or 75%. The five reports for which our duplicate detection tool provided a correct recommendation are marked with an asterisk (*) in Fig 4.11 and are grouped on the left.

For all reports for which our tool provided a correct recommendation (reports #1 to #5), the treatment group was able to more accurately determine an appropriate duplicate. In particular, for report #3, about four times as many treatment group subjects were able to find the previous duplicate report. For report #4, the treatment group was roughly twice as accurate as the control group.

For the remaining five reports, our tool did not provide any correct recom-

mendations for the treatment group subjects. Even though the tool is providing, in each of these cases, seven incorrect recommendations, we see that the treatment subjects were able to find appropriate duplicates or mark a bug appropriately as unique despite having seen these recommendations. The only bug for which the control group was more accurate than the treatment group was bug #8, the only unique bug. None of the control group subjects marked this bug as a duplicate, but two study group subjects marked the report as a duplicate with the recommendations from the tool. For bug #6, no subjects in either the control or the treatment group were able to find an appropriate duplicate. This data shows promise that triagers are able to discern which recommendations are correct and are not overly biased to use one of the recommendations all of the time.

An interesting observation was made when analyzing incorrectly identified reports. Many of the searches by both control and treatment subjects had actually returned one or more correct duplicates, but were missed. This occurred in 61% and 48% of the control and treatment group's incorrect reports. About a third of the control and treatment group's searches (31% and 29% respectively) performed for these reports retrieved at least one correct duplicate. If both groups could have identified the duplicates in their search results, the control and treatment group accuracies could have increased by 27% and 15%, respectively. Determining why the subjects missed these duplicates is left for future work.

4.4.5 Workload

The second hypothesis of interest is whether recommending possible duplicates decreases the workload of a triager. A main contributor to the workload of a triager attempting to find duplicate reports is forming searches over existing repository reports and examining the results. Our tool would reduce triager workload if it reduced the number of searches a triager must perform.

Subjects in both the control and treatment groups used searches to complete the assigned triage tasks. In the control group, subjects were only provided the Bugzilla search engine to locate duplicate reports. In the treatment group, subjects were directed to first use the duplicate recommendation tool and if that did not provide any duplicates, to then consider using the Bugzilla search engine.

Figure 4.12 shows the average number of searches performed across subjects

in each group for each bug report. As before, the bug reports for which our duplicate recommender provided correct recommendations are marked with an asterisk (*). A distinct feature in this graph is that the treatment group subjects performed nearly zero searches for three of the five reports, #1, #3 and #4, for which our tool provided appropriate recommendations. The graph also shows that, on average, the subjects in the treatment group performed less searches than the control subjects. These two features of the graph provide evidence to support our hypothesis that providing recommendations for duplicates reduces a triager's workload.

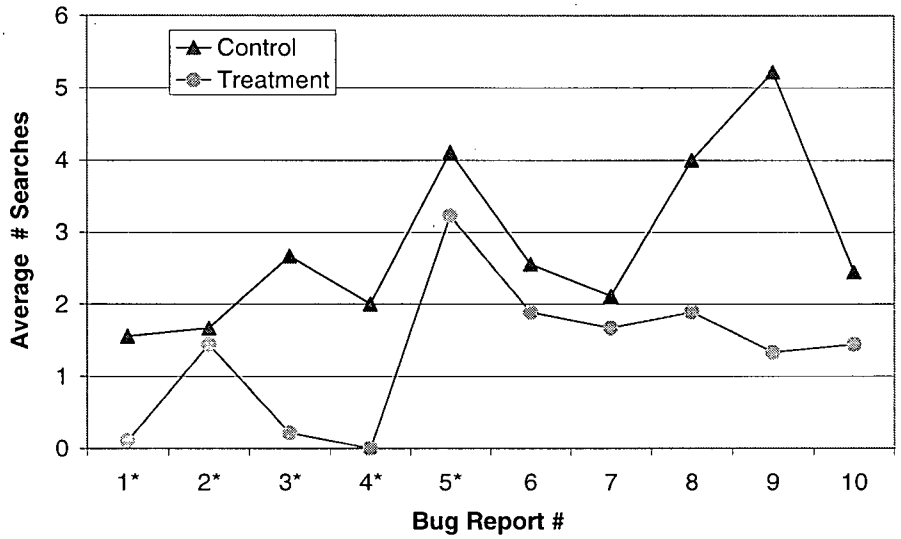

Figure 4.12: Average # Searches Performed on Bug Reports

4.4.6 Time

When human subjects are performing a task under observation, time is a commonly used metric. In our case, we hypothesize that our tool would reduce the time triagers spent trying to identify duplicate reports. We define the time to triage a report for duplicates in our study as the amount of time spent working on the report. We start the timer when a subject first opens the bug report and we stop when either the report is marked as completed or the subject has opened the next bug report. Because some of our subjects did not complete this remote study in one uninterrupted session, we accounted for breaks taken

by a subject by identifying time differences of more than five minutes between two chronologically logged actions. Times exceeding the 5 minute limit were subtracted from the total computed triage time. Our definition of time spent includes the amount of time the subject takes to read the report, to search for duplicates and to mark the duplicates that are found.

Figure 4.13 shows the distribution of the total triage time spent (in seconds) by each subject across all 10 bugs triaged. Each interval represents the range of times starting from the time stated in the preceding interval to the current interval's stated time. For example, the interval labelled with 2000 represents the range of times from 1501 to 2000 seconds.

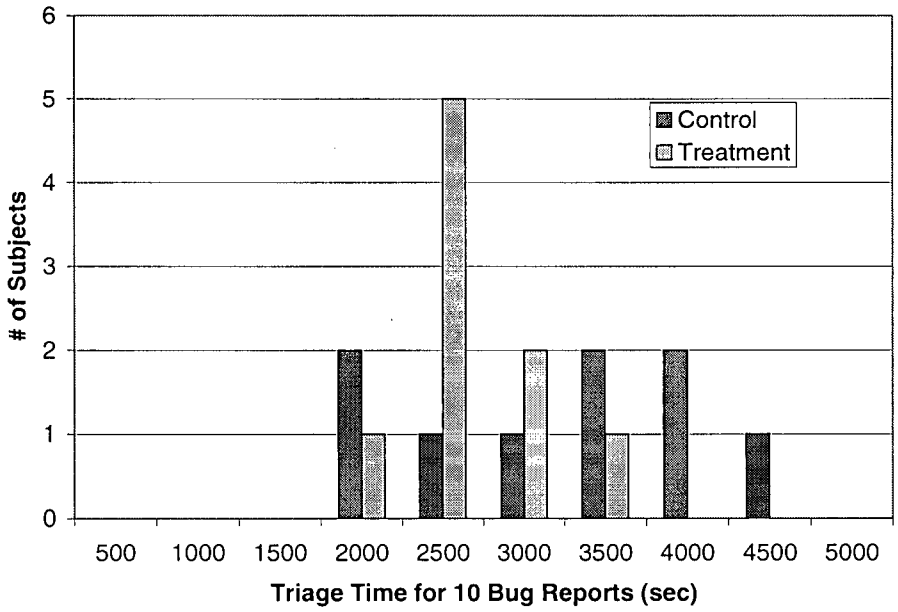

Figure 4.13: Triage Time Distribution

On average, the treatment group spent slightly less time than the control group on the ten reports. Considering that subjects in the treatment group used the recommendations first, and that in half the cases, the provided recommendations did not include a correct answer, it appears that the overhead introduced by perusing the recommendation list is not a liability in terms of the time taken to process reports.

4.5 Threats

A number of choices we made in our study setup threaten the validity of the study results.

4.5.1 A Firefox Study

One could argue that the results of our study do not generalize beyond the Firefox repository. In many ways, the reports in the Firefox repository are representative of other projects. Firefox attracts a wide range of users, who submit a diverse range of bug reports. The scope and size of the project, the diversity of users and developers can also be comparable to other projects. It is also an active project, with new reports being submitted daily.

On the other hand, the Firefox reports are not as technical as some other projects, such as Eclipse or Apache. Firefox reports consists mainly of GUI descriptions, whereas Eclipse reports contain more code snippets and stack traces. Other projects may have bug reports submitted by more technical users or developers and may be written in a more technical fashion that is not easily understood by new bug triagers. Also, projects with a smaller group of users can create repositories with reports of better consistent quality.

Even though the four projects to which we applied our approach resulted in similar performance profiles, a user study may not produce the same results. One area of future work is to carry out user studies on different types of open source projects to see whether the results are similar to our study on Firefox.

4.5.2 Choice of Bugs

The bugs used in the study were chosen to be as representative as possible for reports in the Firefox project. The four training bugs that were chosen ranged from fairly easy to moderate difficulty, allowing subjects to become familiar with Firefox reports and more easily learn how to look for duplicates. The 10 study bugs ranged from easy to difficult. All the reports were about different parts of the Firefox project, ranging from a report about the tabbed browsing feature to a report about Javascript code and XML. The reports covered crashes, unexpected behavior and feature requests to represent the range of reports a Firefox bug triager would see.

Reports were selected to produce a recommendation accuracy just below the optimal duplicate accuracy shown in Fig 4.1 for a threshold value of 0.2. The selection consisted of mainly duplicates, to investigate the effectiveness of our approach on them. The report selection also varied the ranking of correct recommendations, allowing us to see how well triagers were able to find them among the incorrect recommendations.

4.5.3 Previous Triage Experience

Upon completion of the study, subjects were asked to complete a questionnaire about themselves and their experience with the study. When asked if they had triaged bugs reports before in a reasonable capacity (e.g., in a job position or for an open source project), five subjects said yes. Three of these subjects happened to have been placed in the control group and two were in the treatment group. One would expect that having previous bug triage experience would be an advantage in our study, but surprisingly, all of the five subjects had scores comparable to their group's average score. The three experienced control group subjects all have scores of five, where the average was 4.7, and the two treatment group subjects has scores of 6.5 and 7 among the average score of 6.2. These results would indicate that having previous triage experience had little influence on the study results.

4.5.4 Measuring Triage Time

Although differences in reading speed can affect the measured triage time, the metric is still important to show how much overhead is introduced by using the tool. Time differences can also arise depending on how many bug reports are read, the number of searches performed and how thoroughly a subject performed the task. Despite all of these factors, Figure 4.13 shows that both the control and treatment group time distributions are fairly close to each other and appears to indicate that there is some consistency in both the inter-group and intra-group times.

To account for large gaps of time in a subject's recorded history, we removed any time gaps exceeding five minutes from the measured triage time. On further analysis, we see that this heuristic has some merit. It identified 3 subjects who took a break from the study. The duration of these breaks were more

than 25 minutes, representing significant time gaps. On closer examination, all of these breaks occur in between periods of frequent activity, where logged events occurred less than one minute apart. This indicates that this method of identifying time gaps is valid and necessary for obtaining a more accurate measure of triage time.

Chapter 5

Discussion

We discuss factors of the problem domain that complicate duplicate detection and discuss future avenues of research.

5.1 Why Duplicate Detection is Hard

One might ask why no one in the study achieved a score of 10/10 and why our approach has a low precision. There are many possible reasons for this phenomenon. First, bug reports are written in natural language. Some people will inherently describe the same bug using different words. On the other hand, two reports using the same set of words can describe two very different bugs. For example, the Firefox bug reports # 297150 and 269326 from Section 2.2, discussed feature enhancements to add the ability to view the source of a web page and edit it as well. The first report refers to the *page source window* but the second calls it the *View source feature*. These are synonymous with each other, but to our approach, they seem like two different features.

The complexity of a large project with a diverse set of features can also add to the difficulty of finding duplicates. Such a project can create bugs that present themselves in different ways. For example, a 32-bit network API in Firefox limited downloads to 2GB in size, causing both the download manager and FTP client to fail when downloading files larger than 2 GB. This illustrates how one bug, a 32-bit API, can cause other bugs to occur, such as bugs in the GUI, as well as causing multiple different bugs to occurs.

Deciding if a report is a duplicate is inherently subjective, and depends on the bug triager, the reporter of the bug and the project's definition of a duplicate. Our results are based on the assumption that the triager's `UNIQUE` and `DUPLICATE` markings are correct. As such, the results are affected by incorrectly labelled `UNIQUE` and `DUPLICATE` reports, missed duplicates, and incorrectly marked duplicate-of ids.

In addition, when the number of bugs in the repository is large, the ability to distinguish between reports becomes more difficult. In Figures 4.5, 4.6, 4.7 and 4.8 we see the duplicate accuracy for the first few thousand bug reports remains high. However, this drops as more bug reports are added to the model. When a detection system must choose between ten of thousands of reports, this is made even more difficult when the majority of bug reports are either unique or are small groups of duplicates. When there are many bug reports related to a common feature, distinguishing between subtly different reports becomes a challenging task.

In addition to describing a bug, reports also include other noisy features that provide little help in identifying duplicates. Bug reporters sometimes discuss what might be the cause of the problem and suggest how it might be fixed. The description may also indicate which other bug it may be a duplicate of and why they think this bug could be a duplicate. Some reports include a small test case that demonstrates the unexpected behavior, such as a website address, Java code, or an Apache configuration file. Although these test cases can be useful in identifying similar bugs, they can sometimes be quite long and introduce large amounts of noisy text.

5.2 Mistakes made by the Recommender

To fully analyze the reason for low performance of our approach, each recommendation for a report would have to be examined in detail. Due to the time involved in this task, only a sample of bug reports have been looked at. We analyzed a sample of 100 duplicate bug reports from each project where none of the top 7 recommendations were correct. We found that in approximately 40% of the cases, the recommended bug reports were related, affecting the same feature, but were not duplicates. An example is bug report # 297482:

> When scrolling in page using the mouse's wheel or using the keyboard (arrows, PgUp, PgDn) onscroll event is not generated. If I use the scrollbar's buttons or drag the scroller the event is generated. I found this error while developing JavaScript which changes dinamically the position of a HTML element when scrolling or esizing the window.

The 5 closest bug reports, ranked by cosine similarity, all involve scrolling the mouse wheel (Bug Reports # 287533, 212556, 231288, 259294 and 279117). Unfortunately, none are these reports describe the same problem with generating an onscroll event. Another observation is that bug reports about common features tends to have lower duplicate detection accuracy. When there are many bugs about a common feature, attempting to distinguish between them becomes more difficult.

5.3 Improving the Approach

Our work to date has focused on duplicate detection. The overall effectiveness of our approach could be improved by considering the identification of unique reports as these reports make up the largest percentage of the reports in our four datasets. Unfortunately, proving that a report is unique requires showing that no duplicates exist.

The approach could also be improved by analyzing the bug reports of different repositories and looking for features that could be exploited. Event detection systems detect new events in news streams, by identifying the first story that describes a new event. These systems have used properties of news stories to improve detection accuracy, such as patterns of occurrence and named entities (e.g. people or places) [1]. Similar features could be used for bug reports to distinguish between unique and duplicate instances.

To tackle the problems associated with natural language such as synonyms, the use of a thesaurus of domain specific words may help. Creating such a thesaurus would be specific to each project, but constructing one by hand would be too time consuming. One possible approach is to use that of Crouch and Yang [6] to automatically build a thesaurus using the existing bug reports as input. Their approach uses statistical methods and clustering of documents. Once created, the thesaurus could be rebuilt on a periodic basis to keep up to date with changes in the project's language and its usage.

Another area of future work is to use a semantic based approach to duplicate detection. Information extraction has been used to *extract* the relevant information that one is interested in from text, such as which stocks went up today in the business news. In the FASTUS system [9], various linguistic features are extracted from the text, such as tokens, phrases, events and entities. The

system then attempts to fill in a template with this information. An example of a completed template from [9] is shown below:

Incident: ATTACK/BOMBING
Date: 14 Apr 89
Location: El Salvador : San Salvador
Instr: "explosives"
Perp: "guerrillas"
PTarg: "Merino's home"
HTarg: "Merino"

The information in the template is extracted from a news article and is used to fill in the specified fields. A bug report could be semantically analyzed in a similar fashion to extract information relevant to bug reports. Information such as the affected component, error messages and the faulty behavior produced, could be identified from the bug report text. Understanding the bug report at a finer resolution may allow duplicates to be identified more accurately.

Chapter 6

Conclusion

Bug reports play a critical role in an open source project. Bug fixes and enhancements are all stored in the bug repository, a central place where bug triagers, users and developers all interact together to improve the product. The presence of duplicate reports in the repository can gum up this process; for instance, duplicate reports can cause duplicate work, at least at the step of triaging, to be performed and can cause mis-communications, if separate discussions ensue on unidentified duplicates. Current manual processes to prevent duplicates have had limited success on reducing the number of duplicates that exist in open source bug repositories.

We have presented a semi-automated duplicate detection approach to recommend possible duplicate reports to a bug triager. Given a new bug report, previous reports are recommended as duplicates of the current report being investigated. The bug triager can examine the top n recommendations from our detection system to determine if any are actually duplicates. On a data set of Firefox bug reports, our detection approach achieves 29% precision and 50% recall. In a user study where newcomer bug triagers processed ten reports from the Firefox repository, we found our approach can increase the accuracy of identifying duplicates, while reducing the number of searches executed and reducing the time spent triaging.

Bibliography

[1] James Allan, Ron Papka, and Victor Lavrenko. On-line new event detection and tracking. In *Proc. of SIGIR '98*, pages 37–45, 1998.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of ICSE 2006*, 2006.

[3] C. Buckley, C. Cardie, S. Mardis, M. Mitra, D. Pierce, K. Wagstaff, and J. Walz. The smart/empire tipster ir system. In *Proc. of TIPSTER Phase III*, pages 107–121, 1999.

[4] Abdur Chowdhury, Ophir Frieder, David Grossman, and Mary Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[5] Jack G. Conrad, Xi S. Guo, and Cindy P. Schriber. Online duplicate document detection: signature reliability in a dynamic retrieval environment. In *Proc. of CIKM '03*, pages 443–452, 2003.

[6] Carolyn J. Crouch and Bokyung Yang. Experiments in automatic statistical thesaurus construction. In *Proc. SIGIR '92*, pages 77–88, 1992.

[7] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proc. of WCRE '03*, page 90, 2003.

[8] Donna Harman. Relevance feedback revisited. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 1–10, New York, NY, USA, 1992. ACM Press.

[9] Jerry R. Hobbs, Douglas Appelt, Mabry Tyson, John Bear, and David Israel. Sri international: description of the fastus system used for muc-4. In *Proc. MUC4 '92*, pages 268–275, 1992.

[10] A. Martin, T. K. G. Doddington, M. Ordowski, and M. Przybocki. The det curve in assessment of detection task performance. In *Proc. of EuroSpeech'97*, pages 1895–1898, 1997.

[11] M. F. Porter. *An algorithm for suffix stripping.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[12] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[13] Robert J. Sandusky and Les Gasser. Negotiation and the coordination of information and activity in distributed software problem management. In *Proc. of SIGGROUP '05*, pages 187–196, 2005.

[14] Robert J. Sandusky, Les Gasser, and Gabriel Ripoche. Bug report networks: Varieties, strategies, and impacts in an oss development community. In *Proc. of ICSE Workshop on Mining Software Repositories*, 2004.

[15] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[16] Richard B. Segal and Jeffrey O. Kephart. Mailcat: an intelligent assistant for organizing e-mail. In *Proc. of AAAI '99/IAAI '99*, pages 925–926, 1999.

[17] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proc. of ICSE '03*, pages 408–418, 2003.

[18] Yiming Yang, Tom Pierce, and Jaime Carbonell. A study of retrospective and on-line event detection. In *Proc. of SIGIR '98*, pages 28–36, 1998.

Appendix A

BREB Certificate of Approval