2011

Duplicate Defect Detection

Tomi Prifti
West Virginia University

Duplicate Defect Detection

Tomi Prifti

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Bojan Cukic, Ph.D., Chair
Tim Menzies, Ph.D.
Arun Ross, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2011

Keywords: Duplicate detection, Information Retrieval, Bug Repositories

Abstract

Duplicate Defect Detection

Tomi Prifti

Discovering and fixing faults is an unavoidable process in Software Engineering. It is always a good practice to document and organize fault reports. This facilitates the effectiveness of development and maintenance process. Bug Tracking Repositories, such as Bugzilla, are designed to provide fault reporting facilities for developers, testers and users of the system. Allowing anyone to contribute finding and reporting faults has an immediate impact on software quality. However, this benefit comes with one side-effect. Users often file reports that describe the same fault. This increases the triaging time spent by the maintainers. At the same time, important information required to fix the fault is likely to be distributed across different reports.
The objective of this thesis is twofold. First, we want to understand the dynamics of bug report filing for a large, long duration open source project, Firefox. Second, we present a new approach that can reduce the number of duplicate reports. The novel element in the proposed approach is the ability to concentrate the search for duplicates on specific portions of the bug repository. This improves the performance of Information Retrieval techniques and classification runtime of our algorithm. Our system can be deployed as a search tool to help reporters query the repository or it can be adopted to help maintainers detect duplicate reports. In both cases the performance is satisfactory. When tested as a search tool our system is able to detect up to 53% of duplicate reports. The approach adapted for maintainers has a maximum recall rate of 59%.

Acknowledgments

I would like to acknowledge my parents, my fiance and my brother for their support. I would also like to thank my advisor, Dr. Cukic, who believed in me and helped me in completing the master program and my thesis. I would like to acknowledge also Sean Banerjee and Adam Nelson for their contribution in my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Bug Tracking Systems

The role of bug tracking systems in software development is vital. They give users an opportunity to report and describe failure incidents. Software developers benefit from such a communal model of problem reporting when fixing underlying faults. Such problem tracking systems allow users to become “testers”, thus increasing the likelihood that observed failures are accounted for and, eventually, eliminated. This has an immediate impact on the quality of the software. Although the term ”bug” lacks a formal definition, in this context it generalizes the known definitions of software faults (programming errors) and failures (externally observed departures from specified or user expected behavior). Problem reports typically address failure occurrences, leading to fault localization and correction. Occasionally, a bug report may prompt localization and correction of multiple faults. With this understanding, we will continue to use the term ”bug” and adhere with the informal vocabulary dominant in problem (issue, bug) reporting and tracking domain.

The advantages of open bug tracking systems, however, come at a price. Users can file different reports describing the manifestations of the same fault. This gives rise to duplicate bug reports. To distinguish between new and duplicate bug reports, a triager, someone with a good understanding

of the system, is assigned to examine each filed report. He / she renders a decision on whether the new bug report describes a known issue, or a yet unknown, unreported one. The number of duplicate reports in some projects constitutes a considerable percentage of the total number of reports. Thirty percent of bug reports in Mozilla have been recognized as duplicates [7]. This large number of duplicate reports gives the developers no choice but to dedicate a significant effort in triaging, rather than analyzing, debugging and fault resolution. However, most developers consider duplicate reports as an important source for additional information about the bug [10]. Duplicates provide information that may not be present in the initial bug report. For example the initial bug report may contain poor, inadequate or even incorrect information [9]. Duplicate reports may provide information critical for resolution, such as *steps to reproduce* failures, *stack traces*, or better descriptions of failure occurrences.

Why do users submit duplicate reports? Bettenburg et al. [10] noted that *poor search features* of bug tracking systems are often the reason. Problem reporting guidelines, such as the ones for ECLIPSE [2], stress the importance of searching for duplicate reports. However, search features provided by bug tracking systems are of limited scope and do not help users find the correct matches [21]. A number of duplicate reports for MOZILLA start with the statement "Apologies if this is a duplicate" [10]. This typically indicates that the reporter performed a few queries and could not find the matching bug report.

The objective of this thesis is twofold. First, we want to understand the dynamics of bug report filing for a large, long duration open source project, Firefox. Second, we propose and evaluate an approach that helps users and triagers find existing duplicate reports. In the first phase of our experiment we test an approach that can be used to improve search features of a bug repository and help users find existing bugs before filing a new report. Users will search the repository by providing short text descriptions of the fault. A short list of existing reports containing the most "actively" reported bugs will be presented to the user for examination. If the user finds the report that matches his / her observations in the suggested list, filing of a duplicate report may be avoided.

The recommended procedure would be to provide additional information to the existing report in form of comments. This approach would move some of the triaging duties from the developers to the users. By examining a short list of suggested reports, users become more knowledgeable of some of the most actively reported failures and performance incidents. Even if the bug they are looking for is not included in the suggested list, users would still have the chance to read about some existing problems and may avoid filing a duplicate report in the future. In the second phase of our experiments we expand our approach to help triagers resolve the status of a new report. At this point in time we assume that the user has filed a new report and triagers are trying to determine if they are dealing with a new fault or the report describes an existing fault already reported.

1.2 Statement of Thesis

Our approach can help users and triagers find duplicate reports with minimal initial information and execution runtime:

- Users and triagers can search for existing duplicates by entering short descriptions of the problem.
- Our approach can help triagers resolve the status of new incoming reports.
- Our approach is unique in restricting the search pool of bugs in the repository and having recall rates that are better than existing state of the art algorithms. The algorithm runtime is not affected by the size of the bug repository.

1.3 Contributions of this Thesis

To the best of our knowledge no related work in the field has offered a detailed analysis of all defect reports from a long term project such as Firefox. Understanding the dynamic of bug report filing gives important clues as to where the search for duplicates should be concentrated. Most of the related work propose state of the art algorithms that perform extremely well in specific time periods. These works test their approaches in limited data that in most cases do not exceed more than three or four months of reports from a project. Their true detection capabilities may be questionable if deployed in production repositories. We test our approach using all the data from the Firefox project. This includes reports from a period of more than ten years. An important finding from our analysis is that the time interval between consecutive duplicate reports can be used as a sole predictor to limit the search space. Incorporating this finding in our algorithm improves the classification accuracy significantly and reduces runtime when applied to large report databases such as Firefox.

1.4 Structure of this Thesis

The remaining chapters of this thesis are structured as follows:

- Chapter 2 describes in brief some Natural Language Processing (NLP) techniques. NLP techniques are numerous and it is difficult to include all of them in a single chapter. We will concentrate mainly on NLP techniques used in our approach as well as some techniques that are widely used in the field of duplicate detection. We also provide a short overview of the latest related work in this field.

- Chapter 3 provides a detailed description of the data from the Firefox project. In our analysis and experiments we include all valid bug reports from the project's initiation until the most recent release. We investigate some of the reasons that we think may be related to duplicate

report submission. At the same time we attempt to understand the dynamics of bug report filing for a long duration project such as Firefox. We will make use of one important finding from this analysis to boost the performance of our algorithm, as described in Chapter 4.

- Chapter 4 gives a detailed description of our approach. This chapter is divided into two parts. In the first part we propose a technique that could be used to improve the search features of a open source bug repository and remove a considerable workload from the triagers. The second part presents a technique that is designed to assist triagers in finding duplicate reports. Both techniques are built on the grounds of important empirical findings from Chapter 3.

- Chapter 5 provides a short discussion on the difficulties characterizing duplicate detection in open source bug repositories. We conclude this chapter by stating some assumptions that underly the validity of our findings.

- Chapter 6 summarizes the results of our work. In addition we propose some new research directions that can strengthen the validity of our work and improve performance.

Chapter 2

Related Work

2.1 Information Retrieval and Natural Language Processing

Information Retrieval (IR) generally refers to the process of extracting interesting information from document repositories. The information in these documents is mainly textual, expressed in Natural Language (NL). IR is a variation of Data Mining with the main difference being the structure of data that is analyzed. If Data Mining tools are designed to process structured data, IR deals with unstructured data such as full-text documents, emails, HTML etc. As most information nowadays (80 %) is stored in form of NL, IR is believed to play an important role in organizing and retrieving useful information from this vast amount of unstructured knowledge [31]. Companies and organizations could largely benefit from efficient IR tools to optimize their business goals.

In its simplest form, the IR problem could be defined as an ad-hoc retrieval problem [27]. The user searches a repository of documents by issuing a short query describing the information he/she is looking for. The system returns a list of documents that might contain the information desired by the user. The common IR algorithms cannot directly process the text documents in their original form. A preprocessing step is needed to convert the documents in a more manageable representation. This preprocessing steps output a structured representation of documents that constitute the

input for similarity calculation. Based on Manning and Schulze [23], the preprocessing steps are as follows:

- tokenization
- stemming
- stop words removal
- document representation
- similarity calculation

Similarity calculation vaguely can be considered as a preprocessing step. However, following the definitions of preprocessing steps on related literature [23], we will consider similarity calculation part of the document preprocessing stage. The following subsections provide a short overview of each preprocessing steps listed above.

2.1.1 Tokenization

Tokenization is the process of converting a string of characters into a string of tokens. The most common approach is called “bag-of-words” where each word in the document constitutes a token. The “bag-of-words” approach does not store any order information of words in a sentence. Each token (word) goes through a set of ”cleaning” procedures that include: converting capitals letters to lowercase, removing punctuation, brackets and other grammatical symbols.

2.1.2 Stemming

Stemming reduces words to their root. Words may be written in different grammatical forms, but still have the same information. Stemming allows for a more precise comparison between text documents by creating a common form for words that have the same root (e.g “sends” and “sending” both reduce to the same word “send”).

2.1.3 Stop Words Removal

Many common words do not carry any useful information and hence may not be of any help when computing document similarity. This process removes stop words such as “a” and “and”, that are common for many text descriptions and do not contribute to the semantics.

2.1.4 Document Representation

Upon the completion of the preprocessing steps described above, each document is represented in a more manageable form. In our experiments, each report is represented by a *feature vector*. A feature is a dimension in the feature space and the document is a vector in the feature space. Each unique word represents a dimension in the feature space. Different weighting schemes can be applied to assign a value for each feature. The simplest is the *binary* method in which the feature weight is either 0 or 1 - depending on the presence of the corresponding word in the document. A more complex weighting scheme is *TF-IDF* [14]. Each word w in a document d, is given a weight according to formula (2.1).

$$TFIDF(w_i, d_j) = TF(w_i, d_j) * \log(\frac{Dsum}{Dw_i}) \tag{2.1}$$

In formula (2.1), $TF(w_i, d_j)$ is the term frequency of word w_i in document d_j. More precisely, $TF(w_i, d_j)$ is the frequency of the i-th index word appearing in the document or query. $\log \frac{Dsum}{Dw_i}$ is called *inverse document frequency* where *Dsum* is the total number of documents, and D_{wi} is the number of documents that contains the i-th index word. *TFIDF* weighting scheme is assigning a greater weight to those words that frequently appear in a particular document but rarely appear in the rest of the documents. If a word appears in all the documents, *TFIDF* gives a weight of 0 to that word (inverse document frequency will be 0, so the overall weight value will be 0). Words that appear in all or the majority of documents do not contain any useful information that can be used to distinguish between documents. *TFIDF* gives a minimal weight (or 0) to these words. As

a result, these words do not influence the similarity measures.

2.1.5 Similarity Measures

Once documents are transformed in feature vectors and weights are assigned using weighting scheme such as *TFIDF*, the next step is to compute the similarity between two documents (or between the query and a document). The documents can then be ranked according to the similarity score. The highest ranked documents are those most similar to the query (or comparing document).

The most common similarity score is computed by the "dot product" between two documents expressed as feature vectors [15]. If both vectors have been normalized then the dot product represents the cosine of the angle between the two vectors, hence this similarity measure is called "cosine similarity". Formula (2.2) gives the cosine similarity between two vectors *v1* and *v2*.

$$cos(\Theta) = \frac{v_1 \bullet v_2}{\|v_1\| \times \|v_2\|} \tag{2.2}$$

The score ranges from -1, meaning exactly opposite, to 1 meaning exactly the same. Since the weights given by TF/IDF are non-negative, the cosine score in our case ranges from 0 to 1.

Other common similarity measures used are *Dice* and *Jaccard*. In contrast with cosine similarity these two measures "normalize" by avoiding term frequencies altogether. According to Salton, "The choice of a particular similarity measure for a certain application is not prescribed by any theoretical consideration, and is left to the user." [29].

2.1.6 Bug Report Representation

A defect report submitted to Bugzilla repository is composed of free-form text fields and predefined fields. The Summary and the description of the report are in free-form text, whereas fields

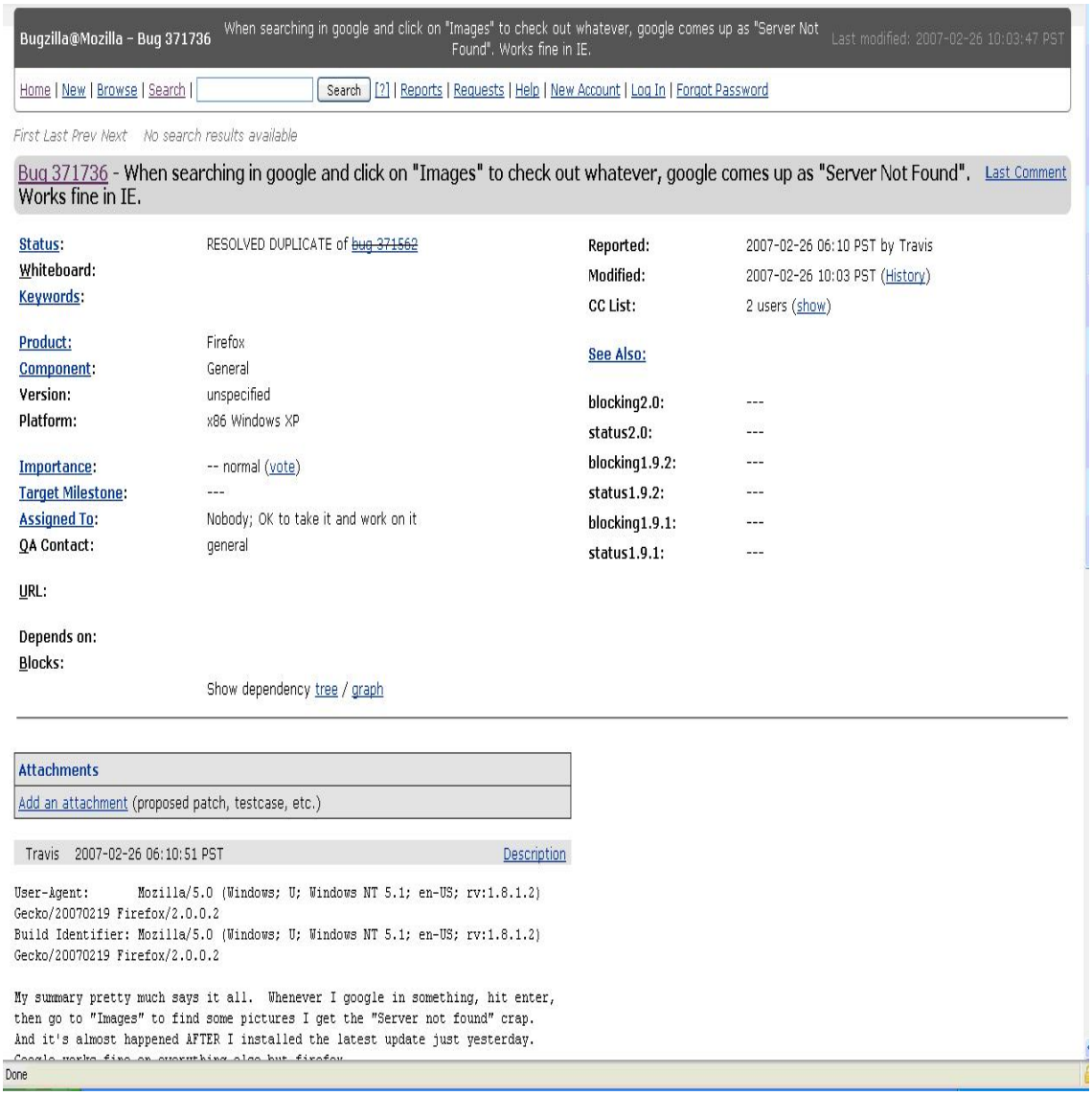

Figure 2.1: Bug Report Fields from Firefox

such as product, component, version and platform are predefined fields. Usually report summaries are short and more concise. On the other hand, descriptions contain more detailed information and are usually longer. Figure 2.1 shows an example of a bug report in Firefox. After examining a bug report, the triager decides if the report constitutes a new defect or is a duplicate of an existing one. If the report is a duplicate, the triager changes the status field to DUPLICATE and indicates the ID of the existing report which the new report is a duplicate of.

Our algorithm is implemented as a Java application. In our experiments we only consider the "summary" and the "description" fields of the bug reports. Each bug report is downloaded and stored as an XML file. AWK is used to parse the files and retrieve the fields that are needed for the experiments including summary and description. Prior to applying similarity measures both "summary" and "description" go through the preprocessing techniques: tokenizing, stemming, stop word removal and feature vector presentation. We make use of libraries from Lucene [17] to perform tokenizing, stemming and stop word removal. Lucene follows the "bag-of-words" approach to tokenize the text and implements the common Porter algorithm for stemming [22]. After the preprocessing steps are completed the documents are transformed to feature vectors. Each vector is composed of multiple pairs of "term" and "weight". In our experiments we test different weighting techniques including *TF/IDF*. The similarity between two reports are computed using the cosine similarity function.

2.2 Triaging Bug Repositories

To the best of our knowledge, this is the first comprehensive effort to analyze long term duplication trends in a large open source project. Furthermore, the proposed use of elapsed time between report submissions as a simple predictor of the upcoming bug duplicate reports is unique. We suggest that the related process that encourages the participation of users in software quality improvement of open source projects can be beneficial. However, it is necessary to consider our findings in the context of the growing body of work in providing automated assistance to bug report triage in open source projects.

The related research in triaging open bug repositories is concentrated in two important directions:

- Improving bug repositories' reporting quality.
- Improving the performance of Information Retrieval techniques.

Both approaches recognize the work overload imposed on the development teams by the arrival of duplicate reports. However, the approach followed by these two streams differ greatly. We will give a short overview on related work for each approach in the following sub-sections. We will also address some of the achievement and shortcomings of each work.

2.2.1 Improving Bug Repositories' reporting quality

Duplicate reports may impose work overload on development teams but at the same time duplicates contain important useful information in addition to the primary report [10]. This additional information can help developers locate and fix the fault faster. Additional duplicates make for the poor, inadequate or even incorrect information of previous reports [9]. Zimmermann et al. [37] proposed four broad directions for the enhancement of bug tracking systems:

- *Tool-centric* approaches aim at facilitating information collection and provision. Enhancements are made to improve features provided by bug tracking systems.

- *Information-centric* enhancements aim at improving the quality of information provided by the developers. Real-time tools can be used to help reporters write better quality reports. If reporters get real-time feedback about the quality of their reports, they may be motivated to provide the extra information required by the developers.

- *Process-centric* improvements focus on administration of activities related to bug fixing. This field incorporates different activities such as automating the step of assigning a bug to a developer, better awareness of the progress made on bug reports or to provide users with an early estimate of when the bugs will be fixed.

- *User-centric* approaches educate and advise users and developers on what information to provide in bug reports and how to collect it.

As mentioned in the introduction of this thesis, poor search features of open bug repositories is the main reason that reporters submit duplicate reports [10]. Users tend to file duplicate reports when they are unable to find previously reported faults. As a result developers spend more time in triaging duties and important information about the fault is spread among different reports. If bug repositories are equipped with better search tools, users will avoid filing new reports but will rather enter additional comments in existing reports. Hooimeijer and Weimer [19] observed that bugs with more comments take less time to fix. In the first part of our experiments we propose an approach that contributes to *tool-centric* and *user-centric* enhancement directions (Referring to Zimmermann list of enhancement given above). By improving bug tracking search features and increasing reporters awareness on some of the open bugs in the system, users can improve their reporting skills by examining a variety of bugs and provide useful information in form of comments to existing reports.

Research work has been conducted in improving the information quality submitted by the users. This work falls under *Information-centric* enhancement direction given by Zimmermann. Users should be guided and trained towards providing useful and quality information in their reports or comments. Bettenburg et al. [9] conclude that the ”steps to reproduce” and ”stack traces” are the most useful information in bug reports. They developed a tool, called CUEZILLA, that measures the quality of bug reports in real time. The tool can recommend additions to bug reports to make their quality better. In addition Breu et al. [11] emphasize the importance of the user in bug fixing process. They state that users can provide a more active role than just reporting bugs. Users can attach useful information in form of comments to existing bugs in order to guide the developers. Their active participation is important for making progress in the bugs they report. These comments have served as motivation for our work.

2.2.2 Improving the performance of Information Retrieval Techniques

Most automated triaging techniques aim at facilitating the work of triagers. The research described in the previous subsection follows a preventive approach to duplicates by improving reporting features of bug repositories. On the other hand, the techniques described in this section apply Information Retrieval techniques to automate the triaging process. At this point it is assumed that the reporter has already submitted a bug report, and a decision on the its status is pending.

Most automated triaging techniques attempt to achieve duplicate detection by comparing the new report with most if not all existing bug reports in the repository [13, 18, 20, 32]. Wang et. al. [32], for example, exploit natural language processing techniques and execution information that may be present in reports to build a list of bug reports that are the most similar to the incoming report. This list is subsequently presented to the triager, who decides on whether the new report is a duplicate or not. Their experiment reflects a small subset of Firefox bug repository and reports a duplicate detection rate as high as *67%-93%*, (*43%-72%* using natural language information alone). We believe that such a high duplicate detection rate may have been attained by an unreasonably small cross section of the Firefox problem reports used in the experiment. Wang et. al. [32] only used the data collected between January 1, 2004 and April 1, 2004, following the release of Firefox version 0.8 on February 6, 2004 (Firefox 0.8 was a technology preview). Their results may have been boosted by short time duration of the study, reflecting the similarity of bug reports over short time span reported in this paper. Our observations, however, are based on the Firefox bug repository data spanning over seven years. It would be interesting to study a synergistic approach which incorporates user analysis of reports, text mining techniques, and additional triage.

Runeson et al. [27] used natural language processing techniques and achieved high recall rates. Their evaluation showed that 2/3 of duplicates can be detected using NLP techniques. However, their data set came from the internal defect reporting system of Sony Ericsson Mobile Communication. Their data set is not as large and as complex as Firefox. In his MS Thesis, Hiew [18] achieved recall rates approaching *50%* using text analysis on Firefox and Eclipse bug reports prior

to 2006.

Another group of related work aims at developing semi-automated triaging systems [8, 12, 13, 20]. Cubranic et. al. [13] apply text categorization techniques to predict the developer that should work on the bug based on the its description. Their approach is based on supervised Basyesian learning where each developer corresponds to a single ”class” and each report is assigned to only one class. The authors builded their data set by selecting all the reports entered into Eclipse's bug tracking system between January 1, 2002 and September 1, 2002. A total of 15,859 reports where selected. Their model was able to correctly classify *30%* of the reports. Anvik et. al. [8] uses a supervised machine learning algorithm that assigns new incoming bug reports to specific developers. They test their approach on data from Firefox and Eclipse projects. The algorithm is considered to be semi-automated because the triager must select the actual developers from a recommended list. Their system achieves precision rates of 57% and 64% for Firefox and Eclipse respectively. The data set used to evaluate their approach consisted of reports filed between September 1, 2004 and May 31, 2005. The training sets included 8,655 reports for Eclipse and 9,752 reports for Firefox.

Jalbert et al. [20] propose a system that automatically classifies duplicate reports as they arrive to save developers time. Their approach uses textual semantic, surface features (version, operating system etc) and a graph clustering technique to predict duplicate status. Their experiments are based on a relatively large sample of 29,000 reports from Mozilla project. The reports span an eight month period from February 2005 to October 2005. Their algorithm is tested in two stages: The first experiment evaluates the performance of the algorithm when applied as a duplicate-detection recall task. They achieve a recall rate of 51% for a suggested list size of 20. The second experiment correspond to a semi-automated system that decides on the status of new incoming bugs. In this scenario, their system is able to reduce development cost by filtering out 8% of duplicate reports while allowing at least one report for each real defect to reach the developers. However, there are few shortcoming influencing the construct validity of their approach. In the first experiment the authors use only duplicate reports to build the corpus of the experiment, thus artificially

eliminating the non-duplicates. In Chapter 3 we will show that the majority of groups contain only a single primary report. Not including these groups introduces a bias to the experiment which may question the efficacy of the model when applied to bug repositories such as Firefox. In the second experiment the performance of the algorithm is evaluated against its ability to filter duplicates. In a survey conducted by Bettenburg et. al. [10], it was found that few developers consider duplicates as a serious problem. In this survey, developers also pointed out that additional information provided by duplicates helps resolving bugs more quickly. This important finding contradicts the efficacy of a filtering system that aims at keeping duplicates away from developers.

The most recent work on the field was conducted by Wu et al. [36]. Their approach first computes field similarities based on Vector Space Model, and then employs Information Entropy to determine field importance. Finally, groups of duplicates are obtained by adopting Hierarchical Clustering. The approach is different from related work in the field which merely draws a magnitude relationship between text field similarity scores. In this approach the authors try to determine the importance values quantitatively. The data set used for the experiment was composed of 3,709 defect patterns extracted from Defect Pattern Repository for Java [3]. The number of reports identified as duplicates was 289. Their approach could detect up to 72% of the actual duplicates. Defect Pattern Repositories are different from open source defect repositories, such as Bugzilla, mainly in that defect reports in the former are specific to a certain project. However, the idea of quantitatively determining the field importance seems promising and its worth testing. To the best of our knowledge no previous work has attempted such approach on open source bug repositories. In Chapter 4 we test our approach with different weighting schemes including Information Entropy.

Last but not least, it is worth mentioning the work by Weiss et. al. [33]. Their techniques aim at predicting the time and effort required to fix a new bug. Given a new bug report, the authors use Information Retrieval techniques to determine a group of similar bugs and use their average time as a prediction. The system could be used as an early effort estimation that can help developers schedule stable releases. The data used to evaluate the approach came from JBoss project managed

by Jira bug repository. Out of 11,185 issues, only 567 reports contained the necessary information to qualify as inputs for their model. An issue could be either a bug, feature request, or task. Their model could beat naive predictions by a factor of four. For issues that where classified as bugs the model could predict the fix time with an error margin of less than an hour.

Chapter 3

Bugzilla Bug Tracking System

3.1 Overview

Bugzilla is a open bug tracking system. It allows individuals or groups of developers to keep track of problem reports related to their software. The term ”open” refers to repositories in which anyone with proper credentials can post a new report or can add comments about an existing issue report. User credentialing policy can be adjusted to best fit the development effort. Subsequently, reports can be filed by developers as well as users of a product. A considerable number of projects use Bugzilla to trace bugs. The official web page [1] provides a list of *961* companies, organizations and projects that use Bugzilla. This list includes well known companies such as Nokia, Facebook, The New York Times etc. Aside from Bugzilla, other common open bug repositories are GNATS [4] and JIRA [5].

Figure 3.1 illustrates the life cycle of reported bugs, as defined in Bugzilla. The status of all reports submitted by developers with ”can confirm” rights, is set to NEW. The reports submitted by users without development credentials are kept in UNCONFIRMED state, which implies a triaging effort to permit the issue to be assigned a NEW state. Once a developer has been assigned to the bug report, the status changes to ASSIGNED. When the bug is fixed, or believed to be fixed, its status

is changed to RESOLVED. Upon further inspection, its status can be changed to VERIFIED or CLOSED. If a bug is not fixed properly its status can be changed to REOPEN. A bug can be resolved in many ways, as indicated by a resolution attribute. The resolution attribute is used to record how the bug was resolved. For example, when a report is found to be a duplicate of an existing bug, the resolution status is set to DUPLICATE. When the bug is fixed in the source code, the resolution attribute is set to FIXED. When the developers are unable to recreate the failure condition, the resolution attribute is set to WORKSFORME. If the developers determine that the fault is impossible to be fixed or the report does not reflect an actual problem, the resolution is set to WONTFIX or INVALID. In all cases, the report status will indicate RESOLVED.

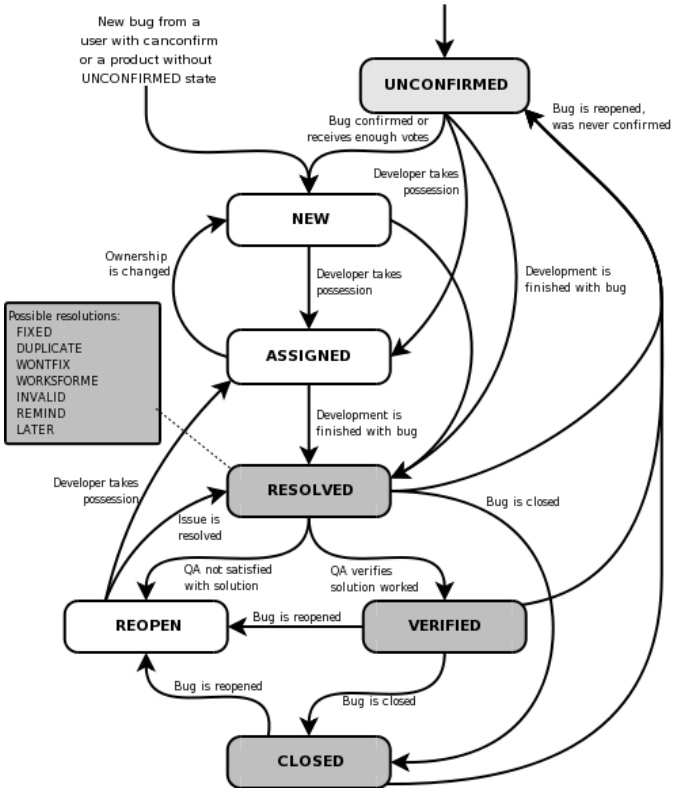

Figure 3.1: Bugzilla Bug Life cycle

3.2 Firefox bug reports

The data for our study comes from Firefox, an open source browser developed by Mozilla Corporation. Development in the Firefox framework consists of a singular Trunk and multiple Branches. The Trunk represents the main code framework that is undergoing continuous, ongoing development. Branches represent forks in the code base that are destined to become end user releases. From its initial release, *1.0*, Firefox has evolved through several major releases: *1.0.5*, *1.5*, *2.0*, *3.0*, *3.5* and the current version *3.6*.

A bug belongs either to one of the releases listed above or to the Trunk. The user or the developer who submits a bug is responsible for specifying the version of Firefox running when failure occurred. In many cases, unfortunately, reporters do not specify the version in the report. In those cases the product version is labeled as UNSPECIFIED.

The total number of bugs in Firefox up to this date is *85,665*. For the purpose of analysis, we filter out *11,077* bugs marked as invalid, leaving a total of *74,588* bug reports. Three bugs, 562340, 562010 and 562295 had a double entry in the database. These bugs where reported recently with a status "UNCONFIRMED". These represent a bug in Bugzilla itself that stores different bug reports with the same ID if their status is UNCONFIRMED. After removing these duplicate entries the total number of bug reports is reduced to *74,585* bugs.

The status of reports that describes existing bugs is set to "DUPLICATE". The number of reports that refer to the same bug form a *"group"* of reports. The total number of reports describing duplicates is *25,507* organized in *10,190* groups. The total number of duplicates does not include the single original bug report in each duplicate group, called primary bug. Developers mark as primary bug, sometimes referred to as master report, only one report in a group of duplicates. Usually the first bug report to arrive is marked as primary, later reports are marked as duplicates, but this is not always the case. When more than two reports describe the same bug "whichever bug report is further in the process of getting fixed is not marked as a duplicate" [26].

Sometimes a duplicate report may have been set as the primary bug of another report. Incorrectly marking of a duplicate as a primary bug of some other reports results in different duplicate groups that describe the same fault. We found that in Firefox bug repository there are *748* bugs that belong in two or more groups. In our analysis we have merged groups that have one or more bug reports in common.

The product version of a primary bug may be different from the versions of duplicates in the same group. Some primary bugs originate in other products (components) used by Firefox. The number of reports that belong to other products is *6,026*. These reports do not belong to the total number (74,585) of bugs mentioned above. In our analysis and experiments we consider only reports that belong to Firefox. We can trace and download all the duplicates for these groups since all of them can be found in Firefox's bug tracking system. On the contrary, it is difficult to download the duplicates in groups whose origin is in products other than Firefox. As stated, these bugs are likely to have additional duplicate reports in different bug reporting databases, reflecting project management practices of their original projects. Duplicate groups of two that have the primary in other products and the only duplicate in Firefox are not considered as groups anymore. In our experiments we consider these groups as having only a single primary bug.

After merging the groups with common bugs and ignoring bug reports from other products we are left with a total of *19,480* duplicate reports organized in *6,408* groups of two or more reports. So, in total there are *25,888* reports divided in *6,408* groups. In addition to these groups we have a total of *48,697* stand-alone primary reports that are not part of any group. Table 3.1 summarizes report distribution in Firefox.

3.2.1 Group sizes

Duplicate reports are organized in groups of different sizes. Figure 3.2 shows the distribution of group sizes. The majority of reports reside in single-report groups which contain only one report (primary report) and no duplicates. Groups of two reports count for the majority of groups

Table 3.1: Bug Distribution in Firefox

Total Number of Bugs	85,665	100%
Number of Bugs (no Invalid)	74,585	87% of Total
Number of Duplicate Bugs	25,888	30.2% of Total
Total Number of groups	6,408	
Number of stand-alone primaries	48,697	56.8% of Total

with duplicates in Firefox. These groups contain only the primary report and a duplicate. As we will discuss later, the prevalence of two-report groups may be attributed to the observation that duplicate reports are often submitted by the same user who submitted the primary report.

Table 3.2 shows the number of groups with sizes one to five and the corresponding percentage of reports. Small groups account to *63%* of the total number of duplicates. In Firefox, we found *60* different group sizes. The overwhelming majority (*90%*) of duplicates resides in small groups of *2 - 16* reports.

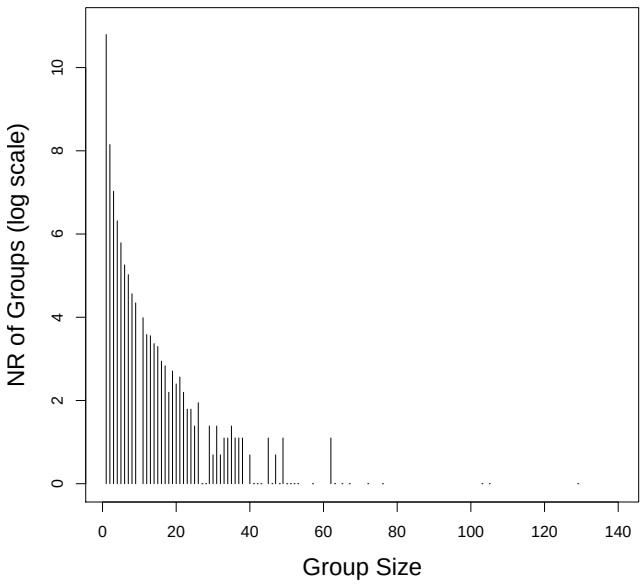

Figure 3.2: Group size distribution. The number of groups on Y axis is presented on *log e* scale.

Table 3.2: Group Size Distribution in Firefox

Group Frequency	Number of Groups	Total Number of Duplicates	% of Reports
Groups of 1	48,697	48,697	65%
Groups of 2	3,464	6,928	9%
Groups of 3	1,129	3,369	4.5%
Groups of 4	554	2,216	3%
Groups of 5	327	1,635	2.2%
Total		62,845	84%

3.2.2 Time interval between consecutive reports

The analysis of group sizes shows that the majority of duplicate reports is organized in small groups with two to sixteen duplicates. In this section we analyze the time interval between the submission of consecutive duplicate reports within a group. In this analysis we consider all *6,408* groups of duplicates that contain more than one report.

Figure 3.3 depicts the time intervals between the first and second report. It is interesting to note that a considerable number of groups receive the second duplicate report no later than *24* hours after the original report was filed. Table 3.3 describes the accumulation of duplicate reports over a period of six days. The first column represents the two consecutive reports. For example, *1-2* means ”the first and the second report”. The first row lists the elapsed time in days, starting with *0*, which indicates ”the same day” submission. The cumulative percentages represent the proportion of groups that received the next duplicate after a specific time interval. For example, *21%* of the groups (column *2*, row *3*) received the second report the same day the first report was filed. *21%* of the groups (row *3*, column *4*) received the second report no later than *2* days after the original report was filed.

The results from table 3.3 can be viewed as the basis for probabilistic prediction when the next duplicate is to be filed. In simple terms, we can say that there is a *16%* chance in Firefox that a bug will receive the first duplicate report on the same day, or a *21%* chance that a bug will receive

the first duplicate within *2* days.

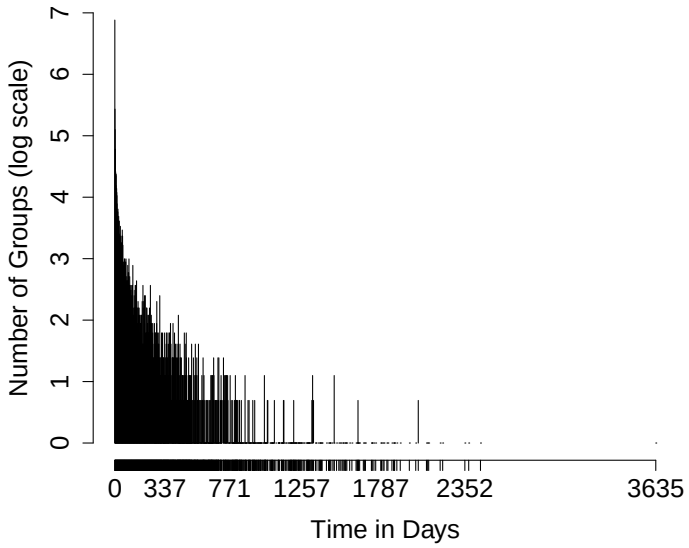

Figure 3.3: Elapsed time between first and second report. The number of groups on Y axis is presented on *log e* scale.

Table 3.3: Time Interval Between Consecutive Reports As a Cumulative Percentage of All Groups

Consecutive Reports	Days passed						
	0	1	2	3	4	5	6
1-2	16%	19%	21%	23%	25%	26%	28%
2-3	13%	16%	18%	20%	23%	25%	26%
3-4	11%	15%	18%	20%	22%	24%	25%
4-5	12%	16%	20%	22%	24%	26%	28%

Figure 3.3 also indicates that the time interval between the first and the second report varies greatly. The majority of bugs receives the first duplicate during the first few days (or months) from the original report. However, a considerable number of bugs received the first duplicate surprisingly late, some several years after the original report. A single bug, shown at the far right end of the x-axis, was reported twice with the duplicate filed *3,635* days (*10* years) after

the original. The first report of this bug was submitted during the early releases of Firefox. The user complained of not having the option to search the exact word in a web page. For example searching results for the word “cat” would include words such as “catch”. A different user filed a duplicate report *10* years later, obviously in frustration. The second report states that “...this is very basic searching function after all”. The initial report was assigned Minor severity. This could be a case of developers ignoring some minor bugs. Elimination of resolution delays could result in a considerable reduction in the number of duplicates.

3.2.3 Bug Dependency

Sometimes a fault cannot be corrected until another fault receives a resolution. This constitutes a dependency. Analyzing bug dependency allows us to understand whether it actually influences the number of duplicate reports in a group. One would expect that bugs with dependencies may take longer to fix. This delay may cause the submission of additional duplicate reports.

Table 3.4 shows the number of groups with and without dependencies and the total number of duplicates for each group. Groups with dependencies count for *19%* of the total number of groups and *21%* of the total number of duplicates. To understand the source of the difference, we note that the average size of groups with dependencies is *4.52*, while the size of groups with no dependencies is *3.92*. The two means are statistically different with p-value=0.0016 at 95% confidence interval. Therefore, we observe that group dependency does have an impact on the number of duplicates, although not as pronounced as we may have expected.

Table 3.4: Group Dependencies

	Groups	Total Dup.	% Dup.
With Dependencies	1217	5503	21 %
Without Dependencies	5191	20385	79 %
Total	6408	25888	100 %

Both group types, with and without dependencies, follow general group size trends, i.e., du-

plicates form groups of small average size. However, there are interesting differences in the distributions of group sizes depicted in Figures 3.4 and 3.5. The groups with two reports form the majority in both classes. Nevertheless, about *55%* of groups with no dependencies are two-report groups, as opposed to only *48%* for the class with dependencies. Similar differences can be observed in the larger groups, resulting in the different average group sizes calculated earlier. This simplistic analysis indicates that dependencies do have an effect on the total number of duplicates per group and the distribution of group sizes.

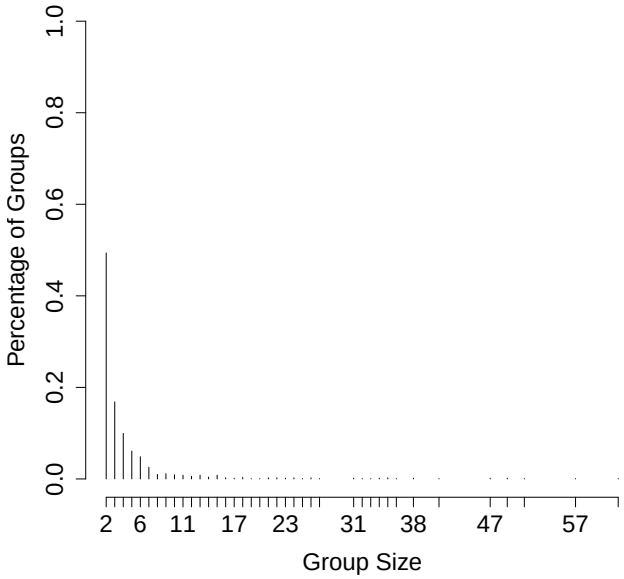

Figure 3.4: Groups with Dependencies: Proportion of duplicate reports per group size.

3.2.4 Bug Severity

Bug severity field of the report describes the perceived impact a bug has on the product. Some reports require immediate attention from the developers, others do not. Problem reports indicating program failures, either as a crash or freeze, may affect the maintenance schedule as well as the development or testing of a new release. The following list describes how bugs are ranked according

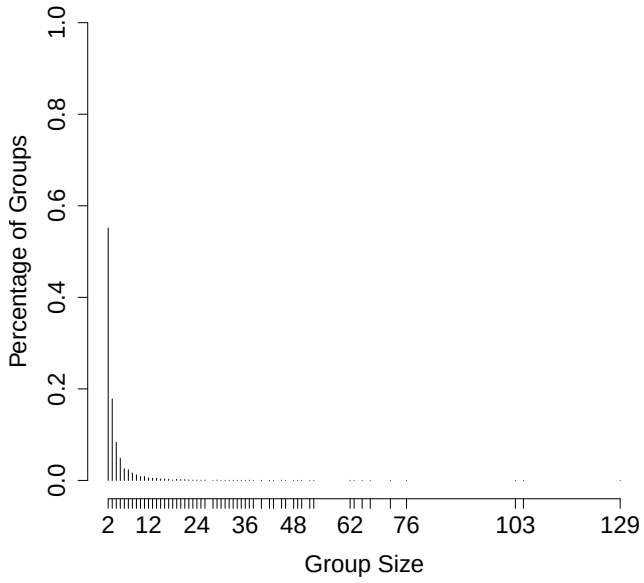

Figure 3.5: Groups without Dependencies: Proportion of duplicate reports per group size.

to their severity starting with the most severe category:

- **Blocker (BL)** Blocks development and/or testing work.
- **Critical (CR)** Crashes, loss of data, severe memory leak
- **Major (MA)** Major loss of function.
- **Normal (NR)** Regular issue, some loss of functionality under specific circumstances.
- **Minor (MI)** Minor loss of function, or other problem where easy work around is available.
- **Trivial (TR)** Cosmetic problem, like misspelled words or misaligned text.
- **Enhancement (EN)** Request for enhancement.

Figure 3.6 shows the number of duplicates and groups per severity category. The majority of bugs are recorded as NORMAL severity and a very small percentage of the bugs belong in

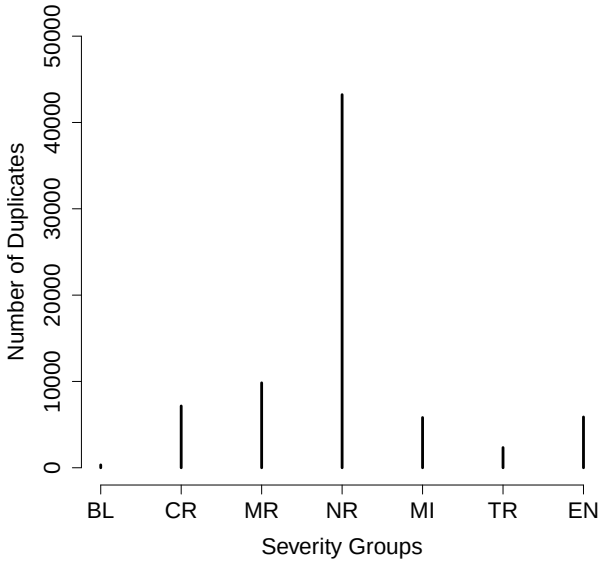

Figure 3.6: The total number of duplicates per severity category.

BLOCKER category. We noticed that in most of the cases duplicates inside a group have different severities. For example the primary report may be of "Critical" severity and the following duplicates may have "Normal" or even "Trivial" severities. Reporters may have different perceptions on severity regarding the same fault. At the same time it is unclear as to what extent are triagers involved in evaluating the severity assigned by the users. It appears that significant effort is not spent no evaluating the severity of reports as assigned by the reporters. In our analysis we consider the severity of the primary report as the representative severity label for the entire group.

Table 3.5 shows the number of duplicates and groups per severity. The last column shows the average number of bugs per group in each category. The difference in averages does not appear to follow any specific trend. All categories of severity have a similar average number of reports per group. Table 3.6 shows the statistical comparison between "Major" severity and the rest of the categories. Groups with "Major" severities tend to have a greater average number of duplicates per group, however, this is not the case when compared to "Minor" category. Both categories, "Major" and "Minor" averages are not statistically different from each other. This brings us to the

conclusion that in Firefox, bug severity does not have a clear influence on the number of duplicate reports.

Table 3.5: Number of Duplicates per Severity Category

Severity	**Duplicates**	**Groups**	**AVG / Group**
Blocker	242	66	3.6
Critical	2,034	544	3.7
Major	3,800	786	4.8
Normal	13,718	3,518	4.0
Minor	2,416	532	4.5
Trivial	747	194	3.9
Enhancement	2,931	768	3.8
Total	25,888	6,408	

Table 3.6: Statistical Comparison of means between different severity groups

Severity	Major
Blocker*	p=0.009 t=2.65
Critical*	p=0.0006 t=3.42
Normal*	p=0.001 t=3.31
Minor	p=0.45 t=0.8
Trivial*	p=0.015 t=2.44
Enhancement*	p=0.0009 t=3.32

*** indicates statistical difference at 95% confidence interval**

3.2.5 Intentional resubmission of duplicate reports

Bettenburg et. al. [10] list some of the main reasons why users submit duplicate reports:

- *Lazy and inexperience users*. Users simply don't spend time searching for duplicates or do not have sufficient experienced with bug tracking systems to undertake a search.
- *Poor search features*. Bug tracking systems such as BUGZILLA do not offer adequate search features to assist reporters in their search for existing bugs.

- *Multiple failure, one defect*. Sometimes it is not obvious that more than one failure is activated by the same fault.

- *Intentional resubmission*. Some users intentionally resubmit a bug out of frustration due to the prior bug not being addressed.

- *Accidental resubmissions*. A number of duplicates is created as a result of users accidentally clicking the submit button multiple times.

We decided to analyze the scale of intentional resubmission in our data. Multiple reports in a group submitted by the same user(s) are considered to be intentionally resubmitted. If a user feels he / she needs to add extra information regarding a bug, the new information can be appended as a comment in the primary report. Typically, software users get frustrated that a bug they reported is not being fixed. As a result, they submit a new bug report hoping it will attract the attention of maintainers and developers.

From the total number of *25,888* duplicates in Firefox *1,139* can be considered intentional duplicates. We also call intentional duplicates redundant reports. If such reporting behavior can be eliminated, the total number of duplicates would decrease by almost 5%. Figure 3.7 shows the number of redundant duplicate reports per group size.

The majority of redundant reports originated from intentional resubmission belong to small groups of bugs. Typically, the second or the third bug report is submitted by the same user that originally reported the same bug. One of the ways to reduce such behavior from reporters would be to update regularly the resolution status on each bug. Users would be able to see that the bug they reported is not forgotten, hopefully eliminating the urge to file a new report.

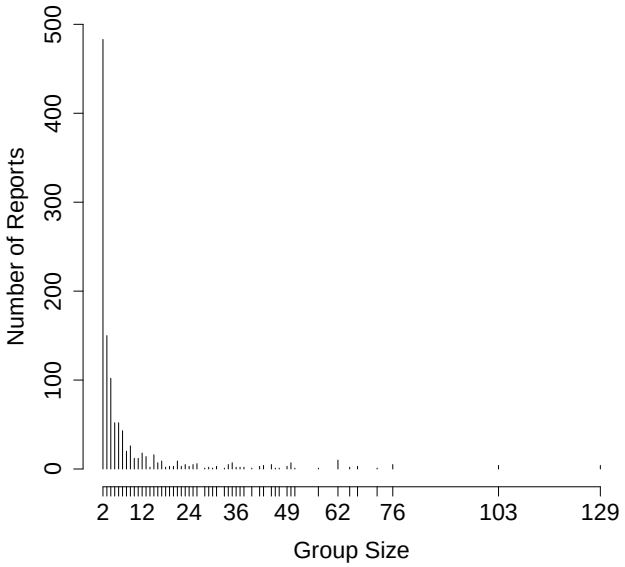

Figure 3.7: The number of redundant reports per group size.

Chapter 4

Duplicate Detection Algorithm

4.1 Triaging in open source bug repositories

Triaging in open source bug repositories is a task that demands significant time and effort. Due to the complexity, the task is typically assigned to a member of the project development staff. Triagers are responsible for examining each new incoming bug report and determine its status. A new report could describe a fault in the system that has never been reported before or could be a duplicate of an existing report. In case when a new report describes a new fault in the system, triagers are responsible for assigning that report to a developer. No matter if a new report is a duplicate or a primary, the triager has to spend a considerable amount of time reading the description and deciding on the status of the report.

Experience from a similar project, ECLIPSE, indicates that *94* new bugs were reported daily between Oct. 2001 - Dec. 2007 [10]. With such a large number of new reports arriving on a daily basis it becomes necessary to assist the work of triagers by automating part of their evaluation process. In Chapter 2 we provided a summary of the research carried out in this field. Different works follow different directions to assist triagers. Some approaches intent to automate the process of assigning a report to a developer. More modest approaches aim at helping triagers find bug

reports or groups that appear similar to the new incoming report. A third research direction aims at improving reporting qualities of bug repositories. No matter the approach followed the common theme is to remove part of the workload from triagers.

4.2 Our Approach

Before presenting our approach we want to stress the main contribution of our work. Almost all of the related work in the field build their data sets by extracting a small subsection of reports from the entire repository. Some of these works test the performance of their algorithms in data sets that do not exceed more than 500 reports. In Chapter 3 we showed that projects repositories, such as Firefox, store 80,000+ reports and this number increases rapidly on a daily basis. Considering only a small section of reports for evaluation introduces a strong bias and doubts the efficacy of ones' approach in real life project repositories. The results obtained by using Information Retrieval techniques may vary greatly depending on the number of text documents considered. On top of that, few studies address the runtime performance of their algorithms if applied to real life repositories. We have to keep in mind that classification runtime is a crucial factor that can't be neglected when building classification systems for real time environments. A system that offers high classification performance but takes to long to build a list of similar reports to present to the triagers is of no practical use.

Our aim is to overcome these important shortcomings in related work. Two very important evaluation criteria characterize our approach.

- We evaluate the classification performance of our algorithm on a data set that includes the entire repository of reports from Firefox.
- Classification performance and runtime is evaluated against the increasing number of reports in the repository.

Including the entire repository of reports in our data set makes our approach more realistic and practical. Before applying Information Retrieval techniques to detect duplicates for a new incoming report we present an approach that will:

- Limit the search space
- Boost the classification performance
- Keep a constant classification runtime

To the best of our knowledge no previous work on the field has experimented with concentrating the search for duplicates in specific portions of the database.

An important finding from Chapter 3 was that the time interval between consecutive reports tends to be short (in most cases one or two days). This finding can be used to concentrate our search for duplicates only on most recent groups of duplicates. In other words, we rank all the groups of duplicates based on the shortest elapsed time between the last report within a group and the arrival time of the new report. Then we conduct our search considering only the top groups in this ranking. This approach can be characterized as a "Sliding-Window" method [25] because for each prediction we first sort the groups chronologically based on the elapsed time and then consider only those groups that are included inside the window. The window is of fixed size so that the number of reports considered for comparison remains constant and so does the runtime. Figure 4.1 depicts our approach. Suppose that the window size is two, the closed groups to the new report based on the elapsed time between the last report and the new report are groups 202 and 203. The search for duplicates is concentrated on reports from these two groups only.

We found that a window size of 2,000 groups, includes the groups that will receive the next duplicate with an accuracy of 95%. This way we limit the search space of reports to an average of 8,000 reports for each prediction. This is a significant reduction considering that we start the experiment with 30,000 reports in the database and this number would increase as the experiment progresses.

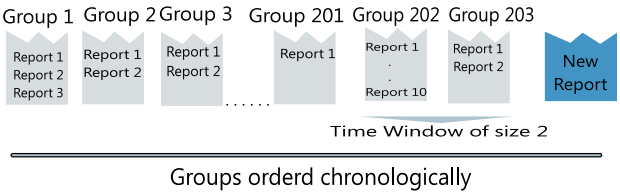

Figure 4.1: Time window.

4.3 Data set and Experimental Setup

In our experiments we used all the reports from Firefox repository. In total, the data set contains *74,585* bug repots clustered in *6,408* groups with more than one report and *48,697* stand-alone primaries. We ordered bug reports chronologically, based on report's filed date. We consider the first *50%* of bug reports as preexistent in the database. Thus, our experiments are conducted on the remaining *50%*. This setup ensures that only the reports received in the past are used to "predict" the content of the new bug reports. We simulated the arrival of each bug based on the reported date. Note that, as the experiment progresses, the proportion of bug reports in the database becomes significantly larger than the number of upcoming reports. For example, in the middle of the experiment *75%* of bug reports are stored in database and *25%* remain in the test set. The experiment runs until *100%* of bug reports are analyzed. If a bug in the test set is a primary bug, i.e., the first time reported, we forward it into the database. This approach simulates the assumption that reporters never misjudge the new bug by reporting it as a variant of an existing one.
Our algorithm runs in two consecutive stages:

1. First we use the "Sliding-Window" method described in the previous section to build an initial list of 2000 groups that are most probable to receive a duplicate report.

2. Then we apply Information Retrieval techniques on the initial list to build a short list that will be presented to triagers or reporters.

We have tested our approach on two different scenarios:

- Scenario 1. Reporters will use our tool to query for existing reports.
- Scenario 2. Triagers will use our tool to find existing duplicates for a new incoming report.

In the first scenario we evaluate the performance of our algorithm as a repository search tool. This approach aim at involving reporters in triaging activities thus reducing the workload of triagers. In the second scenario our tool is used by triagers to detect existing duplicates for a new incoming report.

4.4 Scenario 1. Search tool

In their work Bettenburg et al. [10] noted that bug tracking systems suffer from *poor search features*. The lack of searching capabilities results in users filing duplicate reports. The approach we propose aims at improving the search features of bug tracking system by helping reporters find previously reported bugs. Before filing a report, we propose that users be offered a list of bug reports that are, according to some criteria, likely to describe the problem the user is about to report. Since the user / reporter has not filed the report yet, the only information that we have is his/her Bugzilla login time. Therefore, we build the initial list of bug groups using the window approach and reporter's login time. Then the reporter is prompted to enter a short text briefly describing the problem. To simulate the query entered by the reporter we use the "summary" field of the incoming report. The "summary" filed is used as a query to search the pool of bug groups inside the window. A short list of top bugs is then presented to the reporter. If the reporter recognizes that the bug he/she is about to report already exists, a new duplicate bug report will not be submitted. Instead additional information may be added to the existing bug report in the comments field.
The benefit of this approach would be the reduction of triagers workload. At the same time, this approach encourages the user to add potentially valuable information into the existing reports. In such a way, reporters become more involved in the software quality improvement activities, learn more about the existing problem reports and bugs by examining the items in the list of suggestions.

If in the future the reporter encounters one of the problems she read about before, chances are she may spend more time searching through the existing bug repository. Therefore, the benefits of this approach may compound over time.

4.4.1 Experiments

We conducted four experiments using only the ”summary” field to compare reports. Rather then using all the terms in the ”summary” field, we extracted the top 18 terms ranked using the values given from the weighting scheme. We tested the performance of our algorithm using top terms ranging from one to a maximum of 50. Top terms of 18-20 where found to give better recall rates. The experiments we conducted are described below:

TF/IDF only. In this experiment we compare the document vector representing a new report to every vector that is currently in the database. We did not make use of the ”Sliding-Window” approach to restrict the search space. The ”summary” vectors in the database are weighted using TF/IDF to emphasize rare words. The reports are ranked based on their cosine-similarity scores. The report ranking is used to build the suggested list presented to the user. As the experiment progresses, the number of reports in the database increases. This impacts the runtime since the computational load increases linearly with the number of reports in the database.

Time Window - TF/IDF. In our second experiment, we applied the ”Sliding-Window” to limit the search space. Only the reports within the 2,000 groups of the initial list are considered for comparison. These reports are then weighted using TF/IDF. The scoring and building of the suggested list follows the same procedure as in the first experiment.

Time Window - Group Centroids. In our third experiment we keep using the ”Sliding-Window”, however, the reports from the 2,000 groups are not immediately searched and weighted using TF/IDF. Instead, we build a centroid vector representing each group. The centroid is composed of all unique terms from all reports in the group and the sum of their frequencies in each report. The total frequency of each term is divided by the number of reports in the group.

We present an example to illustrate how we calculate the centroid for each group of duplicates. Suppose that a group contains only three bug reports and we want to compute the centroid of the field ”summary”.

Summary 1 Unable to send email

Summary 2 Sending email is not functional

Summary 3 Can not send email after entering recipient

After the word preprocessing steps the summaries become:

Summary 1 unable send email

Summary 2 send email function

Summary 3 send email after enter recipient

The resulting centroid of the group is:
1.0 send, 0.33 unable, 1.0 mail, 0.33 function, 0.33 after, 0.33 enter, 0.33 recipient

Time Window - Group Centroids - TF/IDF. Our fourth experiments follows the centroid technique described above. However we weight each term in centroids using TF/IDF weighting scheme.

4.4.2 Results

The results of our experiments are presented in Figure 4.2. We tested the prediction performance with different suggested list sizes, ranging from 1 to 20 reports. To evaluate the performance, we use the recall rate of target reports. In other words, if the duplicate report/s of a new incoming report are included in the suggested list then the algorithm succeeds. This evaluation measurement was suggested by Runeson *et al.* [27]. Formula 4.1, below, describes the recall metric:

$$Recall\ Rate = \frac{N_{recalled}}{N_{total}} \tag{4.1}$$

In the above expression 4.1, $N_{recalled}$ refers to the number of duplicate reports for which a representative from the target group (the group of reports that describe the same bug) is in the suggested list. N_{total} refers to the number of duplicate reports whose submission is simulated in our experiment. If the duplicate report in submission belongs to a group whose representative is in the list, we count the classification of the upcoming bug report as correct. As the size of the list increases so does the recall rate. However, we should keep in mind that users may not be willing to examine very long lists of existing bugs, thus creating the need to evaluate the tradeoff between the list size and the recall rate.

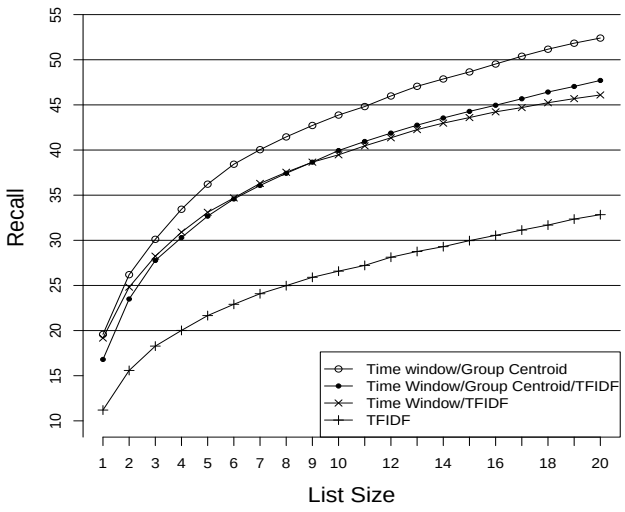

Figure 4.2: Recall rate per list size, using only ”summary” field for comparison

The results of our performance evaluation experiments are presented in figure 4.2. The third experiment using the time window and group centroid yields better recall rates reaching up to 53% with a list size of 20. Using ”Sliding-window” to concentrate the search has a clear impact on the classification performance. The first experiment which searches all the bugs in the database

not only has a lower recall, due to the limitations of Information Retrieval techniques in a large document space, but also the runtime increases with the number of reports in the database. On the contrary, the ”Sliding-Window” used by the other approaches keeps the search time constant, since the number of reports or group centroids remains the same for every new search. The second and the forth experiment have almost the same performance. Using centroids weighted with TF/IDF has a slightly better performance compared to weighting all the reports in the groups with TF/IDF. One important observation from the above experiments is that TF/IDF weighting does not improve the performance. The third experiment, described in the previous section uses as weights for each term the values calculated from the centroid of each group. On the other hand the fourth experiment weights each term in the centroid using TF/IDF. All the centroids drawn from each group are treated as a whole corpus of documents and then weighted using TF/IDF. Jalbert, and Weimer [20] reach the same conclusion regarding the use of TF/IDF weighting. They performed a statistical analysis on a data set of 29,000 defect reports from Mozilla project, trying to determine why the usage of TF/IDF weighting is not helpful in identifying duplicate bug reports. They considered every duplicate bug report and its associated original bug report in turn and calculated the shared-word frequency for the titles and descriptions of that pair. They also calculated the shared-word frequency between each duplicate bug report and the closest non-original report, with ”closest” determined by TF/IDF. Their results showed that TF/IDF is just as likely to relate duplicate-original pairs as it is to relate non-duplicates. There exist a number of different weighting schemes besides TF/IDF. However, most papers, including the work by Jalbert and Weimer, try to empirically derive an optimal weighting scheme on their data set of reports. Our centroid-weighting scheme is simpler and achieves better results than TF/IDF.

Our results demonstrate that by using only the ”Sliding-Window” and the report summary, we can predict duplicate problem reports with an accuracy of up to 53%. Our approach could be easily incorporated as a search feature in bug repositories to help reporter search for defect reports. This implies that the workload associated with the triage of incoming bug reports can be significantly

reduced using a rather simple technique. Furthermore, the information pertinent to the specific problem would end up being concentrated in fewer reports, further assisting system debugging and maintenance activities.

4.5 Scenario 2. Assisting triagers

The experiments described in section 4.2 make use of ”summary” field only. By using ”summary” field to search for duplicates we are investigating the possibility of using our approach as a search feature in bug repositories. At this point users have not filed a report but are searching if the report they are about to file has already been submitted by some other user. In this section we move a step further and assume that the user has already filed a new report. Now it is the duty of the triager to validate and assign the status to this new report. The experiments performed in this section are designed to assist triagers the same way the experiments in section 4.4 were designed to help users find existing reports. The difference is that now we have extra free-text information that the users have entered in the ”description” field of the report.

4.5.1 Combining summary and description scores

To obtain the similarity between two reports we need to combine the scores of ”summary” and ”description” fields. The similarities of these two fields are computed using the approach followed in the third experiment of section 4.2 (time-window and group centroids). The most straight forward way is to sum the similarities of each field weighted by the field importance according to the formula (4.2).

$$Sim = Sim_{summary} \times W_{summary} + Sim_{description} \times W_{description} \tag{4.2}$$

In formula (4.2), *Sim* is the total similarity score expressed as a linear combination of scores

obtained from ”summary” and ”description” fields. $W_{summary}$ *and* $W_{description}$ correspond to the weights that we assign to summary and description scores respectively. In the following sections we experiment with different schemes of assigning the weights.

4.5.2 Magnitude relationship of weights

Different works [8, 18, 20, 32] simply draw a magnitude relationship between summary and description scores. ”Summary” field is recognized to be more useful as users can only enter a limited text that must be descriptive in their problem description. In the ”description” field users tend to be more detailed in describing the problem. In this section we perform two experiments. The first one treats similarity and description scores as equally important. The second experiment gives double weight to the summary score. This way we are treating summary similarities as more important than description similarities. Figure 4.3 shows the recall rates for different list sizes.

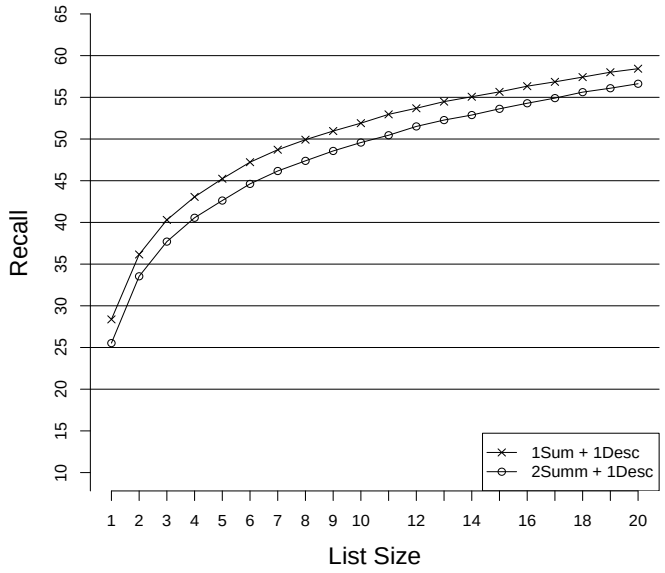

Figure 4.3: Recall rate per list size. Using weighted “description” and “summary”.

The results from figure 4.3 show that weighting summary and description with equal weights

yields better recall rates. Using equal weights we get 2-3% better recall rates for all list sizes. For a suggested list size of 20 the recall rate is **59%**. The above experiments indicates that in our data set "summary" and "description" fields appear to be equally important.

4.5.3 Using Information Theory to obtain the weights

The approach for assigning the weights presented in this section is based on Information Theory (or Entropy) [34,36]. The weights calculated are more granular and may vary between predictions. In other words, the weights are not statically assigned but very based on the groups selected from the "Sliding-Window" approach.

The main idea is that the more diverse the content a field are, the more powerful that field is in detecting duplicates. For example, if the contents of "summary" fields are more diverse than the contents of "description" fields, then the similarity score of "summary" should have a greater weight. So, it is reasonable to weight each score by the degree of diversity of the respective fields. The weight for each field is defined by the formula (4.3).

$$W_i = \frac{\sum_{j=1}^{N}(1 - H(a_{ij}))}{N} = 1 - \frac{\sum_{j=1}^{N}(H(a_{ij}))}{N} \tag{4.3}$$

In formula (4.3), N is the number of words appearing in all instances of the *i-th* field. In our case we have two fields: "summary" and "description". The total number of instances for each field is 2000 since we are using the fixed size "Sliding-Window" and group centroids for each comparison. a_{ij} is a word appearing in one instance of i-th field, and H(a_{ij}) is the information entropy of a_{ij} within the context of i-th field, defined by formula (4.4).

$$H(a_{ij}) = -\sum_{k=1}^{m}(p_{jk} \log_m p_{jk}) \tag{4.4}$$

In formula (4.4), m is the total number of instances of i-th field (in our approach we have 2000 instances), and p_{jk} is the probability that a_{ij} appears in k-th instance of the i-th field. Note that if

a word a_{ij} appears in all instances of a certain field, p_{jk} will be *1/m* constantly and as a result it gets a maximal entropy of 1. However, this word carries no useful information since it could not help distinguish between different documents. So the contribution of this word in calculating the overall weight of the field, in formula (4.3), is **0**.

Since in our approach we use the "Sliding-Window" to limit the search space, the weights for each prediction vary depending on the information entropy of reports inside the window. This makes our approach more flexible in comparison with the static assignment of the weights. Figure 4.4 shows the recall rate using Information Entropy compared to the magnitude assignment of weights.

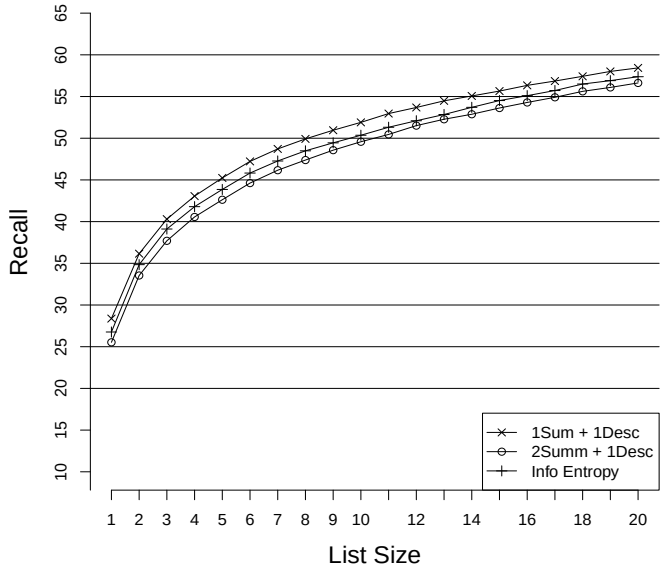

Figure 4.4: Recall rate per list size. Using Information Entropy

The results from figure 4.4 show that assigning weights using Information Entropy of the fields performs slightly better than giving double weight to "summary" field. Using Information Entropy to find the weights does not perform better than considering summary and description scores of equal importance.

Wang et. al. [32] experimented with different weighting values. Their results show that different

weighting schemes do not have a significant impact on the overall recall rates. Jalbert et. al. [20] consider summary and description scores of equal importance without repeating their experiments with different weights.

In our experiments we compared the magnitude assignment of weights with a machine learning approach based on Information Entropy. Results show that equal weights for "summary" and "description" scores yields slightly better results. Yet, the differences in recall rates are not very significant. Information Entropy approach does not outperform equal weight assignment. On top of that it introduces computational overhead which affect classification runtime.

4.6 Recall rate vs Time

The underlying strength of our approach is the usage of "Sliding-Window" approach to limit the search space. As a result the performance of information retrieval techniques is improved significantly. However, one might wonder how is the recall rate of our approach affected by time and the increasing number of reports in the repository? To answer this question we ran the experiment described in section 4.5 (equal weights for summary and description fields), and monitored the recall rate change for each report classified. The test portion of our experiment covers a time span of almost four years, from 2007 to 2010. We compare our approach with the simple TF/IDF experiment that uses no "Sliding-Window" and searches all the reports in repository. Figure 4.5 shows the recall rate as the experiment progresses.

It is logical at the very beginning of the experiment to have high changes in recall values since we calculate recall based on the correctly found duplicates over the total number of predictions computed at that moment. For example, if we correctly classify the first duplicate in the test set, then the recall at that particular moment will be 100%, since the total number of predictions is one and the number of correctly classified duplicates is one. That's why we can observe those fluctuations at the beginning of the experiment.

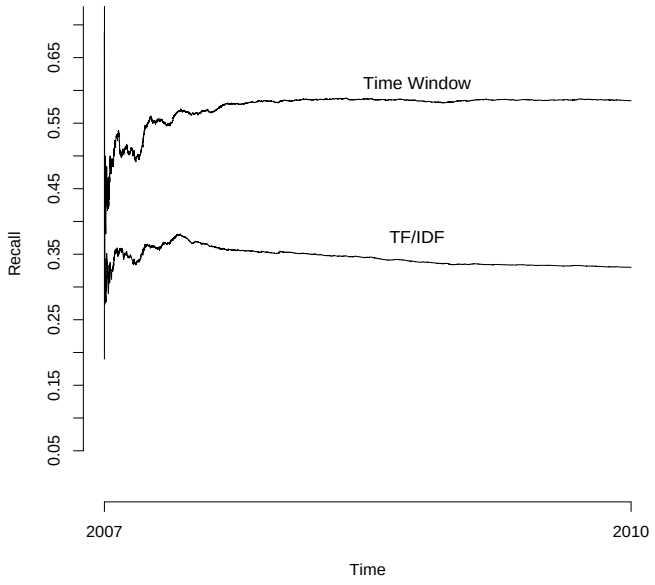

Figure 4.5: Recall rate vs. Time

The most important point to notice is that the recall rate of our approach remains stable for the rest of the experiment and is not influenced by time. Even though the number of reports in the repository increases, the performance of the algorithm does not decrease. This shows that the ”Sliding-Window” approach is persistent with repositories, such as Bugzilla, that receive a considerable number of defect reports on a daily basis. On the contrary, the recall rate of TF/IDF experiment, which considers all the reports in repository, decreases with time. From the above figure we can notice a slight but gradual decrease in recall rate as the number of report in the repository increases with time.

One of the drawbacks of related work in the field is that the performance of algorithms is not evaluated over time. Open Bug Repositories, such as Bugzilla, receive hundreds of new reports every day. It is possible that common Information Retrieval techniques may decrease in performance as the number of reports considered for comparison increases rapidly. This is a strong indication that some sort of guidance in search needs to take place. Our algorithm makes use of ”Sliding-

Window" derived from the empirical study in Chapter 3 to guide the search and keep the number of groups considered for each classification limited to 2000.

4.7 Classification Runtime vs Time

One major benefit resulting from our approach is the fast classification runtime. This classification runtime remains almost constant and is not influenced by the number of reports in the repository. Since the size of the window remains constant at 2,000 groups, the number of reports considered for each classification varies from a minimum of 8,000 to no more than 10,000. This results in a constant runtime as the experiment progresses.

Another factor that influences the runtime is the centroid approach and the avoidance of using TF/IDF for term weighting. As shown in section 4.4, using TF/IDF does not increase performance and it introduces computational overhead. Figure 4.6 shows the classification runtime of our algorithm compared to the runtime of TF/IDF experiment without "Sliding-Window".

The classification runtimes are presented in form of boxplots. The boxplot shows the minimum and the maximum observations represented by the vertical lines on each side of the boxplot. The bold line inside each box represents the median runtime value, and the boxes on the right and left of the median show the lower and upper quartile respectively. The empty circles represent the outliers. In case the of the "Sliding-Window" experiment, although the number of groups for each classification is 2000, the total number of reports varies from 8,000 to 10,000. As a result the classification runtime is not the same for all predictions. The presence of outliers may be attributed to the following reasons.

- Worst case scenarios during sorting and ranking algorithms
- The use of Java to implement the algorithms has its own drawbacks when it comes to memory management. It is possible that during the experiment the Garbage Collection mechanism

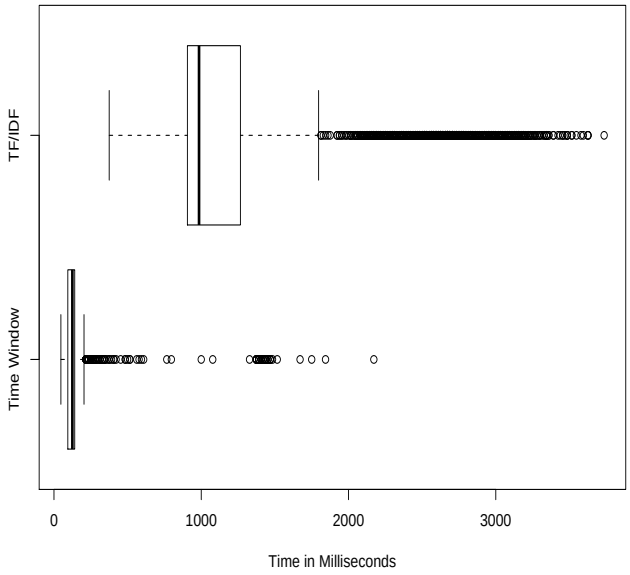

Figure 4.6: Classification Runtime for ”Sliding-Window” and TF/IDF experiments using a 2.13 GHz machine

may kick-in to free memory from unused objects. This may delay the classification runtime of specific predictions resulting in the outliers shown above.

In both experiments we measure only the classification runtime. The preprocessing steps needed to prepare the reports are the same for all algorithms. The median runtime of the ”Sliding-Window” experiment is at least 8 times faster than that of TF/IDF experiment. The median runtimes are *125* and *985* milliseconds for ”Sliding-Window” and TF/IDF respectively. Note that the runtime for TF/IDF is influenced by the increasing number of reports. As the size of the repository increases, so does the classification runtime for TF/IDF experiment. Using the ”Sliding-Window” to guide the search makes our approach suitable for repositories that increase in size rapidly.

4.8 Semi-automated Triaging Simulation

In the experiments that we performed in the previous sections the ground truth was known. We knew beforehand the type of bug (”Duplicate” or ”Primary”) and we test the performance of our approach only on duplicates. The primaries in the test set were forwarded in the repository without making any prediction. In this section we will test the performance of our approach as a filtering system. In this case the ground truth is not known. The system has to determine first whether a report is a ”duplicate” or a ”primary”. In the instance when a duplicate is correctly classified, we take a step further to check if the system actually includes a match in the suggested list.

This approach attempts at reducing the workload of the triager by filtering out of the triaging process those bugs that are primary (describe a new fault) bugs. The triager has to perform the triaging process only on reports that are classified as duplicates. Jalbert et. al. [20], experiment on a similar approach, however, the performance of their classifier is measured on the ability of the system to filter out duplicates and allow only the primaries to reach the developers. This approach contradicts the findings of Bettemburg et. al [10] that recognize the importance of duplicate reports as a vital source of information needed to fix bugs faster. One of their empirical results showed that faults with more duplicates reports take less time to fix. Duplicate reports often contain additional information that are not included in the primary reports. This helps developers locate and fix faults faster. Based on these important findings from related work we evaluate our approach on the ability to filter out primaries rather than duplicate reports.

We will refer to duplicate reports correctly labeled as duplicate by our approach as *”True Positives”*. The *”False Positives”* are primary reports incorrectly labeled as duplicates. The term *”correct duplicate”* will be used to describe those duplicates that are correctly labeled as duplicates and the suggested list presented to the triager includes one or more reports from the same group. To measure the performance of our approach we use three common measures [30] in Information Retrieval field:

$$Recall = \frac{\text{\# correct duplicates}}{\text{\# total number of duplicates}} \tag{4.5}$$

$$Precision = \frac{\text{\# correct duplicates}}{\text{\# reports classified as duplicates}} \tag{4.6}$$

$$F\text{-}Measure = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{4.7}$$

Recall measures the number of *correct duplicates*, out of the total number of duplicates in the repository. *Precision* measures the number of *correct duplicates*, out of the total number of reports classified a duplicates. *F-Measure* is used to measure the tradeoff between precision and recall.
The decision on whether a new report is a ”duplicate” or ”primary” is made based on a predefined score threshold. The textual similarity score is computed between a new incoming report and all the group centroids inside the ”Sliding-Window”. The highest score is selected and compared against the threshold. If the highest score is above the threshold then the new report is classified as a duplicate, otherwise it is filtered out as a primary report.
We tested our approach against 19 different threshold values. The threshold values tested range from 0.1 to 1.9. Since our approach sums up the scores of ”summary” and ”description” fields, scores will range between 0 and 2. Table 4.1 shows the respective measures for each threshold. The best balance between precision and recall (F=0.249) is achieved by using a threshold of 1.3. In this case we achieve recall and precision rates of ***0.376*** and ***0.186*** respectively. The ROC (Receiver Operating Characteristic) offers a visual representation (figure 4.7) of the tradeoff between *True Positive* rate and *False Positive* rate. Each point in the curve is labeled with the corresponding threshold value. We are interested in values that lie along the y-axis. These values correspond to having no false positives. Very high threshold values (close to 2) will classify everything as a primary report. Very low threshold values will classify all the reports as duplicates. With very low threshold values (close to 0), we expect to get recall values of 59%, the same recall value achieved

in subsection 4.5.2, yet, precision will decrease to its minimum since we classify every report as a duplicate.

The ROC curve is a standard evaluation method of classification performance. However, ROC curve ignores the cost implication of miss-classifying a report as duplicate or primary. The cost of miss-classifying a primary report as duplicate or vice versa needs to have an impact on the selection of the threshold. Jalbert et. al. [20] are trying to evaluate the performance of their classifier by introducing two symbolic costs: *Triage* and *Miss*. *Triage* represents the cost of triaging a report while *Miss* represents the cost of ignoring a primary report and all its duplicates altogether. As stated at the beginning of this section we feel that this approach is not consistent with the empirical findings of the related work. Their approach does not penalize the system for filtering out a duplicate report. We feel that a more analytical study should be conducted in order to derive adequate costs for report miss-classifications. At the same time a better representation of cost implications should be deployed other than simple ROC curve that ignores such implications. Sacanamboy et. al. [28] propose a methodology for combined analysis of performance and security risks of a system. They recognize the shortcomings of ROC curves and make use of Cost Curves as an alternative description of classifier performance. We intend to adopt our approach in the future to better describe the performance of a semi-automatic triaging system.

Table 4.1: Threshold results

Threshold	True Positives	False Positives	Recall	Precision	F-Measure
0.1	1	0.999958	0.59	0.122	0.201
0.2	1	0.999958	0.59	0.122	0.201
0.3	1	0.999958	0.59	0.122	0.201
0.4	0.999838	0.999916	0.585	0.121	0.201
0.5	0.999838	0.99962	0.585	0.121	0.201
0.6	0.999675	0.998733	0.585	0.121	0.201
0.7	0.9987	0.996579	0.584	0.121	0.202
0.8	0.996588	0.990625	0.585	0.122	0.202
0.9	0.991225	0.972887	0.58	0.123	0.204
1.0	0.966038	0.918662	0.577	0.128	0.210
1.1	0.906727	0.797964	0.551	0.139	0.222
1.2	0.773318	0.603868	0.486	0.157	0.237
1.3	0.570523	0.378225	0.376	0.186	0.249
1.4	0.348554	0.189493	0.248	0.230	0.239
1.5	0.171271	0.0740318	0.134	0.294	0.184
1.6	0.0690608	0.0231006	0.059	0.378	0.103
1.7	0.0320117	0.0062925	0.029	0.529	0.056
1.8	0.0173871	0.000802399	0.017	0.833	0.033
1.9	0.00487488	8.4463e-05	0.005	0.938	0.001

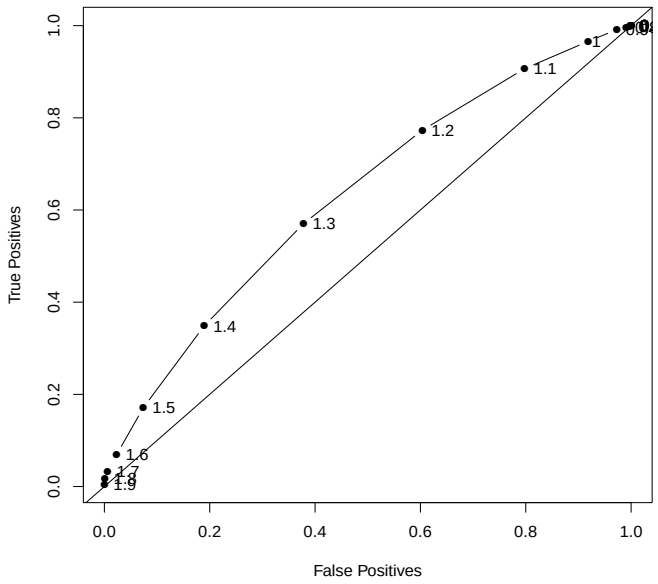

Figure 4.7: ROC curve for the experiment

Chapter 5

Threats to Validity

5.1 Duplicate detection: A difficult problem

In this sections we will discuss a few factors that make duplicate detection in Open Bug Repositories a difficult problem. A recall rate of 59% achieved by our algorithm although outperforms most classifier performance in related work, seems to be relatively low. There are many domain related reasons for such a low performance. We are listing some of them below.

- Bug Reports are written in natural language. Different users may use different phrases to describe the same problem. On the other hand, two reports that use similar words may describe two different things. This is an implication that is ignored during the preprocessing techniques. The document representation (Vector Space and Bag-of-Words) and similarity measures that we use in our approach ignore word ordering and loose some semantic information. Typographical errors by users is another factor that contributes to the complexity of the problem.

  Different similarity measures are designed to cope with some wording problems mentioned above [25]. *Token-Based* similarity methods that we discussed in Chapter 2 and incorporated in our approach, perform well in the presence of swapped or missing words. On the

other hand, *Edit-Based* methods are suitable with typographical errors within words. *Hybrid* measures combine the best of token-based and edit-based methods. However, the computational and runtime complexity should also be consider when choosing between different similarity methods.

- Large number of reports in repositories. Projects such as Firefox or Eclipse accumulate a large number of reports through the years. This has a direct impact on the performance of traditional Information Retrieval techniques. We saw in section 4.7, figure 4.6 that the simple TF/IDF experiment which does not make use of the "Sliding-Window" degrades in performance as more reports are added in repository. Common similarity techniques and weighting schemes are affected by the large amount of reports in repositories. The results from Chapter 4 section 4.4.1 show that weighting terms using TF/IDF, performs worse than the simple centroid approach. Jalbert et. al [20] showed that in the domain of open bug repositories shared-words between reports are likely to increase the similarity between unrelated reports than duplicate reports.

5.2 Threats to Validity

In Chapter 4 we conducted our experiment in two stages: First we evaluated the performance of our system as a search tool for reporters, and in the second stage we adopted our approach to help triagers in detecting duplicates for a new incoming report. In both cases the reporters/triagers are presented with a list of suggested duplicates that has to be examined. The following list points out some assumptions and threats that underly the validity of our findings.

- **Internal Validity.** We assume that triagers always assign a duplicate report to the correct group. However, it is not uncommon in this domain that triagers incorrectly label reports [18]. As a consequence our results may be affected by the number of incorrectly labeled reports.

- **Conclusion Validity.** We assume that reporters are willing to assume a more proactive role in software quality improvement process. Simple bug reporters are driven by good intentions of improving the software quality. They make use of the existing search tools to find out if the report they are about to file has already been reported [10]. However, this may not always be the case and reporters may not be willing to dedicate time examining suggested lists of existing bug reports.

  The given recall rates describe the presence of the matching problem report in the list of suggested reports. In the case of assisting the reporter the recall rate is 53% while for triagers our approach achieves a recall rate of 59% for a list size of 20. These recall rates offer the upper bound on the potential to reduce the number of duplicate reports following the described approach. We assume that users will examine all the suggested reports, correctly detect the match, if one exists, and accept to enhance that report rather than filing the new one. In this case, total number of duplicates would be reduced by the percentage achieved in the recall rates. The same assumption is valid also for triagers. However, not all users may be willing to examine the suggested-list. Impatience, lack of time and a range of other reasons may cause a reporter to ignore the process and file a duplicate report. New or inexperienced users may not be able to identify the bug in the suggested list [10]. On the other hand, some users may think that they found the match in the list of suggestions, although the match does not exist. The same risks affect the triaging process on the developers' side. Even though more experienced, triagers are not immune to classification mistakes. Unfortunately, our experimental framework did not allow us to assess how many original bug reports would eventually end up incorrectly reported as comments in the existing but unrelated reports. This threat reflects the traditional risk of automated or computer supported classification, where the risks of committing Type I and Type II classification errors need to be carefully examined and balanced. To avoid such risks, some triaging efforts, hopefully reduced, may remain necessary for the proposed bug reporting process.

- **External Validity.** As any other empirical study, our findings may not be generalizable outside of the scope of our experimental framework and Firefox problem reporting data set. Although the longevity of the project and the volume of bug reports used in this analysis create a solid empirical foundation, any reported results would need to be reconfirmed using the data sets from other open-source projects.

Chapter 6

Summary and future work

6.1 Summary

In this work we address the issue of duplicate detection in open source bug repositories. Our data set consists of all valid defect reports from Firefox project, covering a period of 10 years from its initial release. In Chapter 3 we analyzed the dynamic trends of bug reporting. The analysis of bug reports collected over 10 years is extensive and provides information on the number of bugs, number of groups, duplicates per group, interval time between consecutive reports, bug dependencies, and relationship between bug severity and duplication. One important finding is that time interval between consecutive reports can be used as a sole predictor on identifying the groups that are more probable of receiving the next duplicate.

In Chapter 4 we build our approach based on the empirical finding from Chapter 3. We introduce the idea of "Sliding-Window" which is used to narrow down the search space for Information Retrieval algorithms. Based on this approach only the 2000 most recent duplicate groups will be considered for each prediction. Each group is represented by a centroid. Rather than weighting each centroid's term using TF/IDF we simply use the resulting weight from the centroid calculation.

We test out approach on two aspects:

- Assisting reporters finding existing duplicates before filing a new report. In this approach we used only the ”summary” field of the reports to simulate a query entered by the reporter. Our approach in this case offered a recall rate of 53%.

- Assisting triager detecting duplicates of new incoming bugs. At this stage we assumed that a new report has been completed and submitted by the user. We adopted our approach to combine the scores of ”summary” and ”description” fields. The resulting recall rate in this case was 59%. We replicated our experiment with different weighting schemes for ”summary” and ”description” fields. Equal weights for summary and description offered a slightly better performance than Information Entropy.

The last section of Chapter 4 evaluated the performance of our approach as a semi-automatic triaging system. This approach is more challenging and requires a better understating of cost implication regarding miss-classifications of reports.
Our approach not only provides satisfactory recall rates, but its performance and runtime are not effected by the continuously increasing number of reports. In section 4.7 we compared our approach to a classical Information Retrieval technique that searches the entire repository for duplicates. We noticed that the recall rate starts to decrease as the size of repository increases. On the contrary, our approach exhibits a steady recall rate and runtime that are not affected by the size of the repository.

6.2 Future work

An important milestone in our future work is to test our approach with data from other projects such as Eclipse [2] or Fedora Core [6]. We need to prove the generality of our approach by applying it to data sets of different projects. For the moment we use only ”summary” and ”description” fields to compare reports. Related works in this field [20, 36], in addition to free text fields, use

predefined fields such as ”version”, ”severity” etc. We plan to include these fields in our similarity function.

When testing our approach as a semi-automatic triaging system we need to include cost related to triaging errors. Using cost curve representation, rather then ROC curves, will provide a better insight of the classifier performance.

At the moment we are comparing documents using vector space model and cosine similarity. Jalbert et. al. [20], in addition to textual similarities of ”summary” and ”description” fields, include also the score from a graph build from existing reports in repository. They use the textual similarities between reports to induce a graph where nodes are reports and edges link reports with similar text. Then a clustering algorithm is applied to form different, probably overlapping clusters. A ”champion” in each cluster (a node with the biggest number of edges) is chosen. For an incoming bug report under consideration, the graph can decide whether a report is a duplicate or a primary based on the proximity to each cluster ”champion”.

We plan to implement a second classification technique that will be combined with textual similarities we have implemented so far. The combined score from these two different techniques will be used to build the suggested list or classify an incoming bug. Common text classification techniques include: Naive Bayes, k-Nearest Neighbors, Neural Networks, Decision Trees etc [35]. Text Classification can be defined as assigning predefined categories to free text documents. In our data set each group of duplicates can be considered as a category. The main concern here is that there will be thousands of categories. There will be at least 2,000 categories if we consider the groups inside the ”Sliding-Window”. The text classification methods mentioned above have been evaluated with data sets that have a small amount of categories (or classes). In our case we are dealing with a multi-class problem.

One powerful classification technique that may perform well in our case is *boosting*. Boosting is one of the most powerful learning ideas introduced in the last twenty years [16]. The motivation for boosting is a procedure that combines the outputs of many ”weak” classifiers and average their

votes to make a classification decision. A common ”weak” classifier used in boosting is decision trees. Decision tree is a well-known paradigm in machine-learning. Common decision tree algorithms include ID3, ASSISTAN, and C4.5. [24]. When applied to text categorization, decisions trees are used to select informative words based on an information gain criteria, and predict categories of each document according to the occurrence of words in the document.

Another tempting reason for considering decision trees and boosting is that they offer an alternative learning/decision method. So far we have used vector space model and associated similarity functions (cosine similarity) to compare different documents. Decision Trees use a different approach for classifying documents. Having two different comparison methods in our approach is an attractive idea since one may complement for the shortcomings of the other.

Bibliography

[1] Bugzilla official website. [Accessed 1-July-2010].

[2] Bugzilla repository for eclipse. [Accessed 1-July-2010].

[3] Code defect pattern repository. [Accessed 1-Sep-2010].

[4] Gnats bug tracking system. [Accessed 1-July-2010].

[5] Jira bug tracking system. [Accessed 1-July-2010].

[6] Red hat bugzilla. [Accessed 1-July-2010].

[7] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM.

[8] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[9] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[10] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful ... really? In *ICSM*, pages 337–345, 2008.

[11] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM.

[12] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1767–1772, New York, NY, USA, 2006. ACM.

[13] Davor Cubranic. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering Knowledge Engineering*, pages 92–97. KSI Press, 2004.

[14] Ronen Feldman and James Sanger. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 2006.

[15] Ed Greengrass. Information retrieval: A survey, 2000.

[16] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2001.

[17] Erik Hatcher and Otis Gospodnetic. *Lucene in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[18] Lyndon Hiew. Assisted detection of duplicate bug reports. Master's thesis, The University of British Columbia, Vancouver, Canada, May, 2006.

[19] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, New York, NY, USA, 2007. ACM.

[20] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52 –61, 2008.

[21] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *In VL/HCC 08: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008.

[22] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on tfidf ranking (poster session). In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, pages 357–359, New York, NY, USA, 2000. ACM.

[23] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, USA, 1999.

[24] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[25] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[26] Sean Richardson. Screening duplicate bugs. [Accessed 3-May-2010].

[27] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE 07: Proceedings of the 29th international conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Mayra Sacanamboy and Bojan Cukic. Combined performance and risk analysis for border management applications. *Dependable Systems and Networks, International Conference on*, 0:403–412, 2010.

[29] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

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

[31] Gurpreet S. Lehal Vishal Gupta. A survey of text mining techniques and applications. In *Journal of Emerging Technologies in Web Intelligence*, volume 1, pages 60 –76, August 2009.

[32] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 461–470, New York, NY, USA, 2008. ACM.

[33] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Qian Wu and Qianxiang Wang. Natural language processing based detection of duplicate defect patterns. *Computer Software and Applications Conference Workshops*, 0:220–225, 2010.

[35] Yiming Yang. An evaluation of statistical approaches to text categorization. *Inf. Retr.*, 1:69–90, May 1999.

[36] Lan Yi, Bing Liu, and Xiaoli Li. Eliminating noisy information in web pages for data mining. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 296–305, New York, NY, USA, 2003. ACM.

[37] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu. Improving bug tracking systems. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 247 –250, May 2009.