2014

An automated framework for problem report triage in large-scale open source problem repositories

Sean K. Banerjee
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
https://researchrepository.wvu.edu/etd/494

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.
An automated framework for problem report triage in large-scale open source problem repositories

Sean K. Banerjee

Dissertation submitted
to the Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in
Computer Science

Bojan Cukic, Ph.D., Chair
Donald Adjeroh, Ph.D.
Mark Culp, Ph.D.
Timothy Menzies, Ph.D.
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2014

Copyright © 2014 Sean K. Banerjee
Abstract

An automated framework for problem report triage in large-scale open source problem repositories

Sean K. Banerjee

Issue tracking systems play a critical role in the management and maintenance of software. Developers and users are allowed to submit reports pertaining to observed problems. A human triager must read through each newly submitted report, and determine if it describes an unreported issue (Primary) or an existing issue (Duplicate). If the quality of the report is deemed to be poor (Incomplete), describing an issue with a different software (Invalid), irreproducible (Worksforme), or beyond the scope of the project (Wontfix) it is annotated with the appropriate label. If the report is deemed to describe a new problem, it is assigned to a developer to work on a solution. In instances when the report is a duplicate, it is assigned the report number associated with the original problem report.

In typical large-scale software systems several hundred problem reports are submitted daily. Thus, the triager faces a daunting task in ensuring that problem reports are quickly annotated with the correct status, and if necessary assigned to a developer. Given the efforts required to triage a problem report, it is desirable to develop automated systems that assist the triager. Existing research in the field has failed to address the problem of automatically determining if a report is Primary or Duplicate. Many efforts made have utilized methodologies that do not scale into the real world, or created artificial datasets that do not adequately model the dynamics of existing repositories.

In this research, we present a fully automated framework that utilizes multiple document similarity measures, summary statistics describing each report and user behavior attributes to determine if the problem at hand is new or duplicate. The framework relies on making as few assumptions as possible on the data in order to reflect the dynamics of a repository. If a problem is deemed to be duplicate, a multi-label classification framework is applied to select the 20 most likely original reports. In order to determine feasibility of the framework, three large-scale datasets from Eclipse (363,770 problem reports), OpenOffice (124,476 problem reports) and Firefox (111,205 problem reports) are used to validate the approach. Our results show that document similarity, user and specific attributes can be employed to differentiate between primary and duplicate reports, with an in-class recall of around 70%. Furthermore, we show that while a silver bullet approach does not exist for determining the correct primary for a duplicate report, a fusion scheme relying on multi-label classification can be used to effectively classify duplicates using simple document classification techniques. Unlike existing research, our results are scalable to the full size of the large-scale datasets.
Dedicated to Natasha (inhabitant of a pale blue dot located in Galactic Sector ZZ9 Plural Z Alpha), my “scientist and atheist” wife, my partner, my best friend, my confidante, and my one true love in this universe (and all potential multiverses).
Acknowledgments

I wish to acknowledge the following people for their personal and professional contributions to this research and dissertation.

My advisor, Dr. Bojan Cukic, for allowing me to have the freedom to explore my own research ideas and not stifling the independence that should be developed during the course of the PhD.

Dr. Donald Adjeroh for teaching the String Algorithms course in Fall 2006, a course that provided the knowledge in sequence matching techniques, which in turn led to my research direction.

Dr. Mark Culp and Dr. Ken Ryan for allowing me to gain access to the Department of Statistics Computing Lab which enabled us to process the seemingly massive datasets in much more finite time.

Dr. Katerina Goseva-Popstojanova and Dr. Tim Menzies for their inputs on the research approaches, and ensuring I had critically evaluated my approach without any bias.

Dr. Roy Nutter for being a mentor, and providing an unbiased perspective into higher education throughout my entire 14 year academic career at West Virginia University.

Jordan Helmick for his contributions to the research, helping run endless processes at Stansbury and for being the “stat guy” by helping the bumbling computer scientists in the lab figure out statistics.

Zahid Syed for making the lab in 215 ERB a relaxed and enjoyable work environment full of many (mis)adventures, and putting in many long nights while we processed the massive datasets.

Dave Shepard and Roy Wendell for the innumerable automotive adventures, and being decade long close friends who understood my desire to seek this terminal degree. Yes, you can finally call me doctor.

To my immediate and extended family.

This dissertation would not have been possible without the love and support of my wife, Natasha. Thank you for listening to my rants and rambles on life and research, while ensuring that I maintained some semblance of sanity during the toughest moments of my degree. This entire work is dedicated to you.
Contents

1 Introduction ........................................ 8
   1.1 Motivation ..................................... 8
   1.2 Goal .......................................... 11
   1.3 Contributions ................................ 12
   1.4 Organization .................................. 14

2 Related Work ...................................... 15
   2.1 Bugzilla Bug Tracking System .................. 15
      2.1.1 Life Cycle of a Bug ......................... 18
   2.2 Automatically Classifying Reports as Primary or Duplicate .... 18
   2.3 Duplicate Problem Report Classification ............... 19
      2.3.1 Word Frequency Based Methods ............. 20
      2.3.2 Word Sequence Based Methods .............. 23
      2.3.3 Topic Modeling Based Methods .............. 23
      2.3.4 Limitations ................................ 24
   2.4 Assigning problem reports ........................ 25
   2.5 Summary ...................................... 27

3 Analysis of Datasets ............................. 28
   3.1 Overall User Behavior .......................... 29
   3.2 Evolution of the Dataset over Time ............... 32
      3.2.1 Total Number of Reports Per Year ........... 33
      3.2.2 Report Types per Year ...................... 34
      3.2.3 Priority and Severity per Year ............. 36
      3.2.4 Changes in Users per Year ................ 38
   3.3 User Type Analysis ................................ 41
   3.4 User Maturity over Time ....................... 46
   3.5 Architectural Analysis .......................... 51
   3.6 Duplicate Group Sizes .......................... 52
   3.7 Discussion .................................... 53

4 The Automated Framework for Classifying Problem Reports as Primary or Duplicate .... 56
   4.1 Dataset Definition ............................... 56
List of Tables

1. Bugzilla Report Attributes ........................................... 16
2. Challenges with existing methods for duplicate detection .......... 24
3. Scalability Challenge in Duplicate Problem Report Classification .. 25
4. Summary of User Behavior ........................................... 30
5. Summary Of Duplicate Submission Behavior .......................... 32
6. Architectural Comparison of Eclipse and Mozilla ................... 52
7. Dataset Definition .................................................... 57
8. Description of the Feature Vector Representing Non-weighted Match Score Statistics ........................................... 63
9. Description of the Feature Vector Representing Weighted Match Score Statistics .................................................... 64
10. Description of the Feature Vector Representing the Content of Each Report ......................................................... 65
11. Description of the Feature Vector Representing Submitter Information ............................................................... 66
12. Proportion of Reports in Raw and Cleaned Datasets .................. 68
13. Summary Results for Primary vs. Duplicate Detection ................ 74
14. Automated Triaging Framework Compared To Naive Threshold Approach ............................................................. 76
15. Time to Manually Triage By Report Type ............................. 78
16. Bug Distribution in Firefox ............................................ 82
17. Group Size Distribution in Firefox ..................................... 82
18. Time Interval Between Consecutive Reports As a Cumulative Percentage of All Groups ................................. 84
19. Worked Example of Cosine Similarity .................................. 92
20. Classifying Report A ................................................... 95
21. Impact of Classification, Component and Product ................... 96
22. Comparison of Results .................................................. 99
23. Multi-Label Classification (Eclipse 2008) ................................ 105
24. Characteristics of the Eclipse, Firefox and Open Office Repositories . 107
26. Baseline Duplicate Report Classification Performance on Firefox w/ Group Centroids ........................................... 109
27. Baseline Duplicate Report Classification Performance on Firefox ............................... 110
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Baseline Duplicate Report Classification Performance on Open Office</td>
<td>110</td>
</tr>
<tr>
<td>29</td>
<td>Proportion of duplicates correctly matched by similarity measures</td>
<td>111</td>
</tr>
<tr>
<td>30</td>
<td>Naive Fusion Scheme Results</td>
<td>112</td>
</tr>
<tr>
<td>31</td>
<td>Duplicate Classification: Effect of List Size on MULAN Recall</td>
<td>124</td>
</tr>
<tr>
<td>32</td>
<td>Fully Automated Triaging - Eclipse</td>
<td>128</td>
</tr>
<tr>
<td>33</td>
<td>Fully Automated Triaging - Firefox</td>
<td>129</td>
</tr>
<tr>
<td>34</td>
<td>Fully Automated Triaging - Open Office</td>
<td>131</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Problem Report Triaging Process</td>
</tr>
<tr>
<td>2</td>
<td>Life Cycle of a Bug</td>
</tr>
<tr>
<td>3</td>
<td>Total Problem Reports Per Year (Eclipse and Mozilla)</td>
</tr>
<tr>
<td>4</td>
<td>Changes in Report Type Over Time</td>
</tr>
<tr>
<td>5</td>
<td>Changes in Priority and Severity Over Time</td>
</tr>
<tr>
<td>6</td>
<td>Changes in Distinct User Behavior Over Time</td>
</tr>
<tr>
<td>7</td>
<td>Changes in User Maturity Over Time</td>
</tr>
<tr>
<td>8</td>
<td>Proportion of Reports by User</td>
</tr>
<tr>
<td>9</td>
<td>Type of Reports by User Type</td>
</tr>
<tr>
<td>10</td>
<td>Priority and Severity by User Type</td>
</tr>
<tr>
<td>11</td>
<td>Report Length By User Type</td>
</tr>
<tr>
<td>12</td>
<td>Evolution of the Super User for Eclipse</td>
</tr>
<tr>
<td>13</td>
<td>Evolution of the Super User for Mozilla</td>
</tr>
<tr>
<td>14</td>
<td>Evolution of the Super User for OpenOffice</td>
</tr>
<tr>
<td>15</td>
<td>Evolution of the Super User (Report Length)</td>
</tr>
<tr>
<td>16</td>
<td>Group Size Comparison (Mozilla and Eclipse)</td>
</tr>
<tr>
<td>17</td>
<td>Primary vs. Duplicate Experiments - Eclipse, Firefox and Open Office</td>
</tr>
<tr>
<td>18</td>
<td>Eclipse Dataset with 500 Trees and 12 Random Features Per Tree</td>
</tr>
<tr>
<td>19</td>
<td>Firefox Dataset with 500 Trees and 20 Random Features Per Tree</td>
</tr>
<tr>
<td>20</td>
<td>Open Office Dataset with 500 Trees and 20 Random Features Per Tree</td>
</tr>
<tr>
<td>21</td>
<td>Eclipse - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences</td>
</tr>
<tr>
<td>22</td>
<td>Firefox - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences</td>
</tr>
<tr>
<td>23</td>
<td>Open Office - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences</td>
</tr>
<tr>
<td>24</td>
<td>Group Size Distribution, y-axis presented in Log 2 scale</td>
</tr>
<tr>
<td>25</td>
<td>Elapsed time between the primary and the first duplicate report. The number of groups on Y axis is presented on log 2 scale</td>
</tr>
<tr>
<td>26</td>
<td>Firefox Recall Rate by List Size</td>
</tr>
<tr>
<td>27</td>
<td>Recall by Experiments (Firefox)</td>
</tr>
<tr>
<td>28</td>
<td>Recall Rates for Different Similarity Measures as Repository Size Grows Using Firefox</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>29</td>
<td>MULAN Experimental Setup</td>
</tr>
<tr>
<td>30</td>
<td>MULAN Results with Eclipse and No Retraining</td>
</tr>
<tr>
<td>31</td>
<td>MULAN Results with Eclipse and Retraining</td>
</tr>
<tr>
<td>32</td>
<td>MULAN Results with Firefox (Group Centroids)</td>
</tr>
<tr>
<td>33</td>
<td>MULAN Results with Firefox and No Retraining</td>
</tr>
<tr>
<td>34</td>
<td>MULAN Results with Firefox and Retraining</td>
</tr>
<tr>
<td>35</td>
<td>MULAN Results with Open Office and No Retraining</td>
</tr>
<tr>
<td>36</td>
<td>MULAN Results with Open Office and Retraining</td>
</tr>
<tr>
<td>37</td>
<td>Impact on Recall as Number of Detectable Methods Changes</td>
</tr>
<tr>
<td>38</td>
<td>Changes in Test Set to Training Set Ratio as Number of Detectable Methods Changes</td>
</tr>
<tr>
<td>39</td>
<td>Impact on Recall Rate as Dataset Size Increases (Eclipse)</td>
</tr>
<tr>
<td>40</td>
<td>Impact on Recall Rate as Dataset Size Increases (Firefox with Group Centroids)</td>
</tr>
<tr>
<td>41</td>
<td>Impact on Recall Rate as Dataset Size Increases (Firefox)</td>
</tr>
<tr>
<td>42</td>
<td>Impact on Recall Rate as Dataset Size Increases (Open Office)</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

The term software “bug” is often used to describe an error, flaw, failure or fault within a computer program. The result causes the program to behave in unexpected ways, or produce incorrect or unexpected results. Software bugs, similar to their biological counterparts, can be deadly. In the 1980s, several patients died due to the Therac-25 radiation therapy machine administering incorrect dosage. In 1996, the European Space Agency Ariane 5 Flight 501 was destroyed 40 seconds after take-off due to a buffer overflow. In April 2014, an OpenSSL vulnerability from 2012 had affected the credentials and private data of consumers across many companies, ranging from Amazon to GitHub.

Developing software without bugs is a utopian dream. Releasing “0-defect”, that is bug free, software is resource, time and cost intensive. As a result issue tracking systems, often called bug repositories, play a critical role in the software management and maintenance process by allowing users and developers to take a proactive role in reporting observed faults and failures within the system. These issue tracking systems are living eco-systems with human users interacting with each other by utilizing natural language to describe the problems they encounter. In general, one may expect that all problem reports within a repository describe real issues. However, since the bug reporting systems allow anyone, with proper credentials, to submit a problem report, it is likely that more than one person will report on the same issue. A later report on the same issue is referred to as a duplicate problem report, with the original report describing the issue being called a primary. Moreover, it is also likely that a user may submit reports of inferior quality (Incomplete), reports that describe problems that cannot be reproduced (Worksforme), reports associated with systems other than the one in question (Invalid), or reports describing issues that are beyond the scope of the project (Wontfix).

When a new problem report is submitted, a member of the development team (called a triager) must ascertain the final outcome of the report by reading through the report. In large-scale, diverse, open source repositories, both developers and regular users alike submit problem reports. The clarity of the language depends on the expertise of the user and their native language. Problem reports are often vague and ambiguous. For example report 915 in Mozilla represents a typical problem report. The short title states:
“(col-align-inherit) implement inheritance of alignment attributes from columns

(align, valign, char, charoff, (lang, dir))”,

while the longer description merely says:

“something about a missing colframe...”.

It is difficult to surmise the issue by evaluating these two pieces of information. Thus, the human triager often must use very limited information to determine the final outcome of the issue. In this case, the problem has remained open and spawned over 78 duplicates as the development team continues working on a fix. If the triager perceives that the newly submitted report describes an existing issue (Duplicate), it is tagged with the identification number of the original report. Reports that are deemed to be actual issues are assigned to a developer to fix. In large software repositories, such as RedHat, Eclipse and Mozilla, a team of triagers may encounter several hundred problem reports per day [23]. Thus, a framework that can assist the triager would not only expedite the process, but also reduce the potential for error.

The language found in a software issue tracking system is vastly different from the language we as humans are used to. Not only are problem reports written in human readable language, but they also contain machine language. We define machine language as the words and phrases used to describe the software system and associated technical content pertaining to the problem report. Each software system has its own unique language. The syntax of the machine language itself may often contain words and phrases that are found in the natural human language. However, the meanings of such words are vastly different. For example, the word “eclipse” in English is used to define the effect of one object overshadowing another. In the context of software, “Eclipse” is a proper noun used to define a development environment. It is often easy to presume that the challenge of triaging problem reports is an information retrieval problem, however, the very nature of the language found in problem reports requires the development of new techniques to address the limitations of standard approaches. While extensive research exists in the domain of natural language processing and information retrieval, our understanding of the marriage between human and machine language is limited.

The emergence of open source software repositories has allowed researchers to formulate solutions to the many problems that plague issue tracking systems. Some of the active areas of research in this domain are: duplicate problem report classification and developer assignment. In the former, each known duplicate is annotated
with the identifier of the correct original report. In the latter, the system determines which developer is best equipped to address the problem. One research challenge that has remained unsolved is determining whether an incoming report is primary or duplicate.

Existing research has consistently applied traditional information retrieval techniques without asking the foremost question: “Is there something fundamentally different about problem reporting systems?” The answer to this question can be found by exploring the deeper facets of each problem reporting system. In this thesis, we analyze three datasets in order to investigate this question. Moreover, it is assumed that an oracle knows when a problem report is new or a duplicate and the research problem to address is one of determining which primary report each duplicate is associated with. This assumption is naive at best, and given the nature of problem repositories, fails to address the larger problem of classifying problem reports as new or duplicate. To date very little research has been conducted in solving the fundamental problem of automatically classifying problem reports as new or duplicate. One can only classify a duplicate report only when it has been determined whether the report is new or duplicate. The mixture of human and machine language in problem reports makes it difficult to apply pure linguistic or information retrieval techniques. Work in [1] attempted to automate this process by using traditional information retrieval techniques. However, the results did not offer much hope in creating a framework that could perform as well as the human. Work in [55] utilized the MeeGo dataset to automatically classify problem reports, however the application of cross validation is implausible in the real world as training cannot be performed with problem reports that have not been submitted. Figure 1 illustrates the basic process that an automated (or human) system must undertake with each newly submitted report.

Due to resource constraints, researchers are often forced to develop and test methodologies on smaller datasets. This creates a research conundrum; the methodologies may appear to work well on a fraction of the dataset while failing when applied to a larger dataset from the same project [5, 8]. For research to be appealing to industry, it is necessary to explore methodologies on large and complex datasets, to assure practitioners that the suggested practices would work on all parts of the dataset as opposed to a very small portion.
1.2 Goal

Existing research in analyzing software repositories makes the assumption that a perfect oracle exists that can determine if an incoming report is primary or duplicate. Only after a report has been determined to be duplicate by a triager can it be passed through the classification system to determine the primary that it matches. Merely performing classification to a duplicate after a triager has identified a duplicate is fundamentally redundant. This is because to determine if a problem report is duplicate, the triager must compare against prior reports, and identify the primary that matches the incoming report. The foremost purpose of the research presented herein is to develop a framework which first classifies a problem report as primary or duplicate, and then provide a suggested list of matches for reports the system classified as duplicate.

Much of the existing research has focused on trying a variety of methods on small subsets of the Eclipse, Mozilla, and Open Office repositories. Most work provides a slight incremental jump in the duplicate classification performance. Yet, one question has remained unanswered—how does the performance of the proposed methods scale as the repository size increases to the extent of the large repositories that are seen today? For example, today there were over 900,000 reports in Mozilla, 450,000 in Eclipse and 1,000,000 in RedHat. Testing each proposed method on large datasets is infeasible and can even be computationally infeasible. This creates a research conundrum as methods that were perceived to be effective on small datasets showed a marked decline in performance on larger datasets. In [5, 8, 11] the authors
demonstrated the challenges faced by developing methods that worked optimally on small subsets of the dataset. The practical implications of using small datasets is that software development companies have no assurance that the proposed methods will work effectively when applied to the entire software repository.

Thus, the second purpose of our research is to develop a framework that is stable over the entire lifespan of the dataset. As a result we chose to utilize the complete repositories for Eclipse, Open Office and Firefox. Problem repositories are living ecosystems comprised of human users submitting problem reports. Through the life span of the software, the behavior of the users can change dramatically. Such changes could be caused by a sudden influx of new users due to the adaptation of the application by the greater community. Or, a sudden outflow of users due to a better product being delivered by a competitor. Novice users can gain expertise over time as they develop a better understanding of the system. In creating a framework that is stable over the lifetime of the software we must include methods that are continually adaptive.

Our goal is two fold. First we present a framework that can work as an automated triaging system while requiring minimal human effort to maintain. Next we propose methods that adapt and scale well to large repositories. In addition, we ask researchers to begin thinking of the differences in human and machine language and their broader implications in research.

1.3 Contributions

The original contribution of this thesis is to first build a complete understanding of large-scale, open source problem repositories by investigating the behavior of both the problem reports and the human submitters. Second, we utilize the body of knowledge gained to develop a systematic framework for determining if a new problem report is primary or duplicate. Currently, this process is performed manually by a triager. Finally, we demonstrate the ineffectiveness of singular document similarity measures and apply a multi-label classification approach to classify duplicate problem reports.

The details of these contributions are as follows:

1. In Chapter 3 we provide an in-depth analysis of three large-scale open source datasets, Mozilla (699,085 reports), Eclipse (363,770 reports) and Open Office (124,476 reports), by exploring various aspects associated with problem
reports and the human submitters. Our original contribution is the first attempt at quantifying the similarity and differences in large open source problem repositories. The three chosen datasets represent a diverse group of projects ranging from web browsers, mail application, development environments and office suites.

2. In Chapter 4 we create an original methodology to automatically classify problem reports as primary or duplicate. The system utilizes document similarity scores, document summary statistics, and user demographic information to create a feature set that is used to train a Random Forest classifier. Incoming reports are automatically classified and the performance measured based on the known ground truth.

3. In Chapter 5 we demonstrate the effectiveness of a group centroid based approach by evaluating the Firefox dataset. Our original contribution is the introduction of a time window scheme that reduces the target search space and generates more effective matches for duplicate problem report classification.

4. In Chapter 6 we demonstrate the weakness of word frequency based methods when compared to sequence based methods for duplicate problem report classification. The effectiveness of the sequence based approach is tested on the Firefox dataset. Our original contribution is the application of sequence based techniques for classifying problem reports.

5. In Chapter 7 we demonstrate that multi-label classification for duplicate problem report classification is superior to any single method. Word frequency and word sequence based measures have inherent flaws that can be overcome by applying a suite of methods. Our original contribution is the application of multi-label classification wherein the system chooses from a pool of methods, as opposed to a single method.

6. In Chapter 8 we provide the fully automated problem classification framework by evaluating the performance on three large-scale open source repositories, namely Eclipse (363,770 reports), Open Office (124,476 reports) and Firefox (111,205 reports).
1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a summary of the related work in automatically classifying duplicate problem reports, automatically assigning a developer to a problem report, and the Bugzilla bug tracking system. Chapter 3 presents a detailed overview of the Mozilla, Eclipse and Open Office datasets, including critical similarities and differences between them. Chapter 4 presents the automated problem report classification system which will be used to determine if a problem report is duplicate or primary. Chapter 5 presents a group centroid word weighting scheme based method for automatically classifying duplicate problem reports. In Chapter 6 we present a longest common subsequence based methodology for classifying duplicate problem reports. Chapter 7 introduces and explores the concept of multi-label classification wherein more than one similarity measure is applied for duplicate problem report classification. Chapter 8 presents the results of the unified framework for automated problem report triaging. We conclude this thesis in chapter 9 by providing a summary and potential future research topics.
2 Related Work

The related literature in open source problem repositories spans four broad areas. The first section presents an overview of the dynamics of the Bugzilla bug tracking system which is used as the issue tracker for all three datasets in our research - Eclipse, Open Office and Firefox. The second section discusses the limited work conducted in automatically classifying problem reports as primary or duplicate. In the third section we discuss the duplicate problem report classification problem using word frequency, sequence based and topic modeling based similarity measures. Finally, we present approaches that addresses the developer assignment problem using machine learning techniques.

2.1 Bugzilla Bug Tracking System

The role of bug tracking systems in software development is vital. Bug tracking systems, otherwise called bug repositories, give users an opportunity to report and describe observed issues with the system. Software developers benefit from a communal model of problem reporting in their attempts to fix the 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. While many bug tracking systems exist, we focus on Bugzilla as it has been used by companies ranging from NASA, Eclipse, Mozilla to RedHat. In this section we briefly describe the Bugzilla bug tracking system and also the life of a bug [29].

Problem reports in Bugzilla generally consist of a short title describing the issue and a detailed summary presenting additional details that can be used by the developer to build a solution. The title is required; however, the summary is optional. Problem reports also consist of a number of additional features, some of which are described in Table 1.

In general the Bug Status and Resolution fields can be unified to a single label - primary or duplicate. A primary describes an issue that has not been reported previously, while duplicate describes an issue that has been already reported. As problem reports are submitted, a triager must utilize the provided textual information to determine whether the report should be tagged as primary and allocated to a developer to be fixed or tagged as a duplicate and annotated with a reference to the original report describing the issue.
<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Submitter</td>
<td>The name of the person who has submitted the problem report</td>
</tr>
<tr>
<td>Date submitted</td>
<td>The date when the problem report was created</td>
</tr>
<tr>
<td>Date modified</td>
<td>The date when the problem report status was last changed</td>
</tr>
<tr>
<td>Classification</td>
<td>Top level categorization for problem reports</td>
</tr>
<tr>
<td>Product</td>
<td>Second level categories used to filter problem reports</td>
</tr>
<tr>
<td>Component</td>
<td>Second level categories used to filter problem reports</td>
</tr>
<tr>
<td>Version</td>
<td>Indicates the software version the problem is associated with</td>
</tr>
<tr>
<td>Operating System</td>
<td>Indicates the operating system the problem is associated with</td>
</tr>
<tr>
<td>Bug Status</td>
<td>Indicates the current state of the problem report. Values range from UNCONFIRMED, NEW, ASSIGNED, etc.</td>
</tr>
<tr>
<td>Resolution</td>
<td>Indicates what was done to the problem report. Takes one of the following values - FIXED, INVALID, WONTFIX, DUPLICATE, WORKSFORME, or INCOMPLETE</td>
</tr>
<tr>
<td>Priority</td>
<td>Indicates how quickly the problem should be fixed</td>
</tr>
<tr>
<td>Severity</td>
<td>Indicates the urgency of the issue. A high severity problem may require development to stop until it is fixed</td>
</tr>
<tr>
<td>Comments</td>
<td>Textual comments provided by other users regarding the problem at hand</td>
</tr>
</tbody>
</table>

Table 1: Bugzilla Report Attributes
Figure 2: Life Cycle of a Bug
2.1.1 Life Cycle of a Bug

Bug repositories are evolving ecosystems, Figure 2 illustrates the life cycle of a typical bug report in the Bugzilla system [30].

As bug reports are submitted they enter the system with either UNCONFIRMED status or a NEW status. A bug report can take a NEW status without first being in the UNCONFIRMED state if the user has “canconfirm” privileges or the product does not have an UNCONFIRMED state. If the bug receives enough votes it can transition from the UNCONFIRMED state to the NEW state, or if it is assigned to a developer it can transition to an ASSIGNED state. If development is completed with the bug then it can enter the RESOLVED state. The RESOLVED state will indicate whether the bug was FIXED, DUPLICATE, WONTFIX, WORKSFORME, or INVALID. The first two resolutions are descriptive. WONTFIX indicates an issue that the development team believes does not need to be addressed, WORKSFORME is a problem that could not be replicated by the development team, and finally INVALID is an issue that is associated with a product other than the one in question. Bug reports can make similar transitions from the NEW state to the ASSIGNED and then RESOLVED state.

Perhaps, of more interest is the fact that when a bug reaches a CLOSED state its life cycle does not end. A bug can easily be reassigned to UNCONFIRMED state and the process can be restarted. Thus, any work with the problem repository is merely done on a snapshot of the actual state. A future snapshot can have different labels assigned to each bug report.

2.2 Automatically Classifying Reports as Primary or Duplicate

Prior attempts at automatically classifying a problem report as new or existing have yielded poor results. In [55] the authors utilized the MeeGo data set and sampled 534 problem reports to determine if they were primary or duplicate. The authors evaluated their approach using 10-fold cross validation, which ignored the fact that reports are submitted into the repository sequentially. Training a classifier with problem reports submitted in the future does not make sense in practice. The authors reported a best overall recall rate of 75%, which was 17% lower than simply assuming all reports in the data set were primaries. Moreover, the authors report a recall rate of 77.9% for duplicates and 61.6% for primaries. The scale of the chosen dataset does not adequately model real world datasets which consist of several hundred thousand
In 2006, Hiew [1] applied a practical threshold based method on similarity scores for identifying duplicate reports in four open source repositories: Firefox, Eclipse, Apache 2.0, and Fedora Core. The data sets used in Hiew’s study were taken from each of the repositories inception until late 2005. It should also be noted that Hiew only considered reports with resolution status being either FIXED, DUPLICATE, or OPEN. Our work relaxes this limitation. Using a top 7 list of candidates, Hiew’s duplicate recall for Eclipse was 20% with a precision of 14%.

In 2014, Lazar [57] utilized complete repositories from Eclipse, Open Office, NetBeans and Mozilla and claim 100% recall for duplicates. The data was cleaned to exclude all reports that were currently open. Removing open bugs is extremely problematic, as open bugs spawn many duplicates. For example, in Mozilla bug 915 would be considered an open bug and yet has spawned 78 duplicates thus far. In a real repository, open reports are ones that have not been assigned to a developer or one where a solution has not been formulated. These reports can continue spawning duplicate reports. The authors also created a pairing system by combining known duplicates and known non duplicates. In a real repository, this pairing information can only be known for a training dataset. Incoming reports cannot be paired unless they are manually inspected. Thus, while the datasets chosen represent the complete datasets, the post processing has created an artificial dataset that cannot model the real repository and cannot be used to classify future reports.

2.3 Duplicate Problem Report Classification

One of the many tasks assigned to the human triager is to determine which primary report each duplicate is associated with. This task can be difficult when dealing with large software repositories with several hundred thousand reports. Moreover, the lack of an effective search tool means submitters are unable to make an informed effort in ensuring they are not willfully submitting duplicate reports. The research in the realm of duplicate problem report classification has primarily focused on using variants of word frequency based methods, word sequence based methods and topic modeling. While machine learning has been used in all three approaches, the differences in approaches are associated with the perspectives taken in measuring document similarity.

The Bugzilla bug tracking system contains its own duplicate detection system. However, this system utilizes a Boolean full text search on the report title. It was
observed across all three datasets, that the median size of the report title was less than 10 words. Obtaining a proper match using such sparse data is challenging, and is clearly reflected by the increased performance in matching duplicates when using both the title and summary of the document.

2.3.1 Word Frequency Based Methods

Word frequency based methods compute similarity between two documents based only on the frequency distribution of words common to them. Naively speaking, two documents are considered similar if the distribution of words between them are closely related. In order to alleviate the false match issue, stop word lists and word weighting schemes are utilized to remove or reduce the impact of unnecessary words. For instance, “the” is the most common word in the English language. As a word it adds little to the ability for a similarity measure to discern between two reports. Thus, a stop word list consists of a set of words that are found most commonly in English and can be used to reduce each document vector into its core terms. Word weighting schemes, on the other hand, provides significance to each word in a document. Lower weights are assigned to words found in all documents, while higher weights are assigned to words found in a set of related documents.

One of the earliest attempts at classifying duplicate problem reports was done by Lyndon Hiew [1]. The datasets used comprised of 4 problem repositories: Firefox, Eclipse, Apache and Fedora. Problem reports were collected from the start of the project until 2005. However, care must be taken when indicating that it was a complete repository, as only problem reports tagged with DUPLICATE, FIXED, and OPEN were included. Thus, large portions of each dataset was excluded, thereby creating an environment which did not adequately simulate a real world situation and one that could potentially boost results. Moreover, the systematic removal of reports makes the approaches challenging to replicate in a real environment where the triager may not know whether an incoming report is DUPLICATE, FIXED or OPEN. In the approach done by Hiew, word weighting was performed with Term Frequency-Inverse Document Frequency (TF-IDF) on each centroid of similar groups of reports. TF-IDF is a word weighting scheme consisting of two parts. Term Frequency (TF) measures how often the word occurs within the specified report. Inverse Document Frequency (IDF) measures how relevant the word is across across all documents and is computed by dividing the total number of documents by the number of documents that contains the word, and then taking the logarithm of the quotient. The cosine
A similarity measure was utilized to measure similarity between an incoming report and each cluster of reports. Modest recall rates were obtained for the top 7 reports in the suggested list. For Firefox this recall was 50%, Eclipse 20%, Apache 32% and Fedora 31%.

Work in duplicate detection has not been limited to open source software problem repositories. Work by Runeson, et-al utilized the Sony Ericsson Mobile Communications dataset [2]. The dataset was first preprocessed to remove stop words, stem all words and finally tokenize words. This normalized set of words for each report was modeled in vector space, where each axis represented a word. Each word was then assigned a weight using a custom term frequency method where the weight for each word is defined as: \( \text{weight}(\text{word}) = 1 + \log_2(\text{tf}(\text{word})) \). Three similarity measures were tried: cosine similarity, dice and jaccard. For a given duplicate bug report, the system would generate the top-k similar bug reports. The proposed methodology was able to correctly identify 40% of the duplicate bug reports when using the cosine similarity measure.

Work by Wang, et-al used the textual information provided in each problem report along with execution information to generate a suggested list of 20 reports [3]. Problem reports were weighted using a traditional TF-IDF scheme. Similarity between documents was computed by using cosine similarity. In order to generate execution information details the authors manually converting any steps to reproduce information to an execution trace. The authors utilized a 3 month (January 1, 2004 to April 1, 2004) window of problem reports consisting of 77 duplicate reports. While the authors demonstrated high recall of nearly 93% when using both natural language and execution information. The recall numbers should be taken with a grain of salt due to the abnormally small dataset centered around a version release. Aside from the small dataset, in many instances an appropriate execution trace may not be available and does not represent an adequate real world scenario.

Jalbert and Weimer extended the work by Runeson by introducing a new word weighting scheme [4]. The new term weighting scheme was defined as follows: \( \text{weight}(\text{word}) = 3 + 2 \times \log_2(\text{tf}(\text{word})) \). Similarity between reports was measured by using cosine similarity. A graph was then generated wherein the nodes reflected problem reports and the edges linked reports with similar text. A clustering algorithm was applied to obtain a set of clustered reports. An overall recall rate of 50% was obtained for matching duplicates to a top-20 list.

Sun, et-al took a different approach to duplicate detection by utilizing a dis-
criminative model [5]. The existing work till this point had focused on using purely
textual similarity measures without any machine learning techniques. Sun, et-al
developed a framework wherein a SVM machine was trained with pairs of actual
duplicates and pairs of non duplicates. The feature vector was constructed by using
54 features obtained from the title, title and summary of each pair of report. The
word weighting schemes were also used for only the title, summary and both title
and summary. The authors tested their system on three projects - Firefox, Eclipse
and Open Office with recall rates of 68%, 67% and 62% respectively.

Our own work in the domain utilized the concept of time windows [7]. Our
research indicated that for Firefox 95% of duplicate reports had a matching report
within the last 2,000 most recently updated groups. By using this notion of 2,000
most recent groups, the search space can be reduced from all prior to reports to
approximately 8,000 reports. This not only increases the response time, but also
reduces the potential for false matches caused by the nature of the English nature.
Weighting of each term in a report was performed by using the group centroid
based weighting scheme. Each group of similar reports is clustered together and
the weight for term \( i \) is computed as follows: 

\[
\text{Weight}_i = \frac{\text{FreqInCluster}_i}{\text{NumberOfReportsInCluster}}.
\]

When experimenting with the Firefox dataset until June 2010 we obtained a recall of
53% using a top-20 list. Moreover, our experimentation noted that a group centroid
based weighting approach was more favorable to the TF-IDF approach.

Sun, et-al extended their earlier work by developing an extension to the BM-
25F similarity measure to work in duplicate detection [8]. BM-25F was originally
developed for short queries, but given the length of prose found in typical bug reports
it cannot be directly applied. One of the advantages of the BM-25F similarity
measure is that it contains a set of tunable parameters that can be used to optimize
the similarity. Thus, unlike cosine similarity it provides more accurate matches.
Moreover, the authors introduce a retrieval function that allows for reports to be
measured for similarity by using categorical features such as product, component,
priority, version, etc. The authors tested their approach on four datasets - Open
Office, Mozilla, Eclipse and Large Eclipse. Recall rates were 70%, 70%, 80% and
70% respectively. We note that the recall rate dropped by 10% when considering 1
year vs. 6 years of Eclipse data.
2.3.2 Word Sequence Based Methods

To date, little research in the domain applied longest common subsequences for detecting duplicate problem reports. We experimented on two datasets, the complete Firefox dataset up to March 2012 and the Eclipse dataset from year 2008 [9]. With Firefox we achieved a best recall of 68% when using a modified longest common subsequence approach. The modified approach measured similarity not just on how well a document matched to another document, but also to the matches within a known group of similar documents. The number of matches in a group would prevent the case when a small document randomly matches to a larger language purely based on random chance. By emphasizing on within group matches we expose groups that closely represent the current test duplicate report. The details of the approach are presented in Section 6. In [10] the authors applied n-grams, as opposed to using a word level match the authors utilize a character level match. Using a character level match will allow for matches between code fragments and English based word phrases. The authors obtain a recall rate of 46% when using 2, 270 duplicate problem reports in the Eclipse dataset.

Prior work on duplicate report classification puts emphasis on word frequency measures using samples of complete repository. We notice, however, the application of string matching techniques for the detection of common usage patterns in execution logs [41]. Other related problems tackled through inexact string matching include the detection of duplicate records in databases [43, 42], name matching [44] and approximate duplicate detection in documents created by optical character recognition [45]. We note that the language found in these applications is far simpler than the free-form English in software problem reporting.

2.3.3 Topic Modeling Based Methods

While sequence based techniques solve one problem, that of context, it introduces an even larger problem, that of individuality. Problem reports are written by humans. In order to be most effective, sequence matching techniques expect two or more humans to use the same subset of words in the same sequence to describe the same issue. Since this may not be the case, topic modeling has emerged as a way to blend the advantages of both frequency and sequence matching techniques.

In topic modeling a system consists of a large corpus of words that can be used to describe each and every submitted problem report [11]. This corpus of words
is often called the vocabulary. Moreover, the system contains a finite set of topics that broadly describe issues found within the system. Each of these topics consist of words obtained from the vocabulary. Thus, a topic $T$ can have 5 words with a relative probability of occurrence based on their frequency. A problem report can be described by one or more topics. In [11] the authors utilized topic modeling with standard information retrieval techniques to obtain recall rates of 80% in Open Office (data from 2008 to 2010) and Mozilla (data from 2010 only) and 85% in Eclipse (data from 2010 only).

However, topic modeling requires that the model is updated continually as the vocabulary changes and the repository size grows. As the repository size grows, the overall vocabulary also grows and it is likely that a duplicate problem report may contain unknown words. The continual update of the vocabulary, and constituent topic word weights is computationally exhaustive.

### 2.3.4 Limitations

There are several limitations to the approaches described in Section 2.3.1, 2.3.2, and 2.3.3. The limitations are summarized in Table 2. One of the goals of our research is to develop a framework that can utilize the benefits of each similarity measure, while at the same time trying to avoid their pitfalls.

<table>
<thead>
<tr>
<th>Method</th>
<th>Word Frequency</th>
<th>Sequence Based</th>
<th>Topic Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td>Large and small documents.</td>
<td>Preserves meaning by maintaining word order.</td>
<td>Addresses context and individuality of author issues by assigning a pool of words to a topic.</td>
</tr>
<tr>
<td></td>
<td>Handles variations in writing style.</td>
<td>Stop word removal is not essential.</td>
<td></td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>Requires stop word removal.</td>
<td>Does not handle individuality in writing styles well.</td>
<td>Requires a dictionary that may not exist.</td>
</tr>
<tr>
<td></td>
<td>Does not preserve meaning of documents.</td>
<td></td>
<td>Building dictionary can be computationally daunting.</td>
</tr>
</tbody>
</table>

Table 2: Challenges with existing methods for duplicate detection
Related literature has also often utilized small samples, with poor scalability. We illustrate this using Table 3, we only include projects utilizing the Eclipse, Mozilla and Open Office datasets as these are the same datasets we utilize. In [3], for example, the authors obtain a recall rate of 93% using 77 duplicate reports from Firefox. When the same method was used on 10% of the Firefox data [5], the recall rate dropped to 53%. Similarly, in [8], the authors noted that by using a larger Eclipse data set (spanning 5+ years), the overall recall rate dropped by 10% when compared to the performance obtained for a single year of data.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Range</th>
<th>Recall</th>
<th>Reports</th>
<th>% of Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>Jan 2008 - Dec 2008</td>
<td>68%</td>
<td>2013</td>
<td>6%</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Start - Dec 2009</td>
<td>46%</td>
<td>2270</td>
<td>6%</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Jan 2008 - Dec 2008</td>
<td>78%</td>
<td>3080</td>
<td>8%</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Start - Dec 2007</td>
<td>71%</td>
<td>27495</td>
<td>100%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Apr 2004 - Jun 2004</td>
<td>93%</td>
<td>77</td>
<td>0.44%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Apr 2002 - Jul 2007</td>
<td>53%</td>
<td>3307</td>
<td>12%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Apr 2002 - Jul 2007</td>
<td>70%</td>
<td>3307</td>
<td>12%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Start - Jun 2010</td>
<td>53%</td>
<td>19480</td>
<td>50%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Start - Mar 2012</td>
<td>68%</td>
<td>25045</td>
<td>50%</td>
</tr>
<tr>
<td>Firefox</td>
<td>Start - Sept 2005</td>
<td>50%</td>
<td>8070</td>
<td>85%</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Jan 2010 - Dec 2010</td>
<td>68%</td>
<td>6925</td>
<td>5%</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Feb 2005 - Oct 2005</td>
<td>51%</td>
<td>8225</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table 3: Scalability Challenge in Duplicate Problem Report Classification

2.4 Assigning problem reports

The task of the bug triager does not end by simply marking a report as primary or duplicate. The triager must then determine which developer is best equipped to deal with the problem. The task of automating the developer assignment process has been performed extensively with machine learning techniques. In this section we explore each approach and provide summary results.

One of the earliest exploration of problem repositories can be found in the early 2000s. Work by Lucca, et-al applied information retrieval techniques to classify bug reports to a broad group of categories. An overall accuracy of 84% was obtained using split and cross sample validation [12]. Podgurski, et-al investigated the use of machine learning techniques to classify and prioritize software faults into specific categories [13].
Exploration of developer assignment began circa 2004 with Cubranic, et-al proposing a text classification based method to semi automate the developer assignment process [14]. The framework was built upon extracting salient keywords from the title and summary of each problem report. These keywords were used in conjunction with the identifier associated with each developer. A Naive Bayes classifier was trained and tested on newly submitted reports. The system was tested on a subset of the Eclipse repository, comprising of reports from January 1, 2002 to September 1, 2002. An overall accuracy of 30% was achieved with this machine learning based method.

One of the challenges of the approach in [14] is the presence of noise in the data. Anvik, et-al extended the initial work by applying a set of filtering schemes [17]. For example, reports that are marked as WONTFIX, WORKSFORME, INVALID need not be assigned to developers. More so, developers are transient and some will eventually retire from a project or stop effectively contributing. Removing such issues would reduce the search space and allow for more effective developer assignment. A suite of three classifiers were tried, SVMs, Naive Bayes and Decision Trees, with SVMs yielding the best results and boosting the overall accuracy to 64% when using the same dataset.

Work by Anvik [18] focused on developing a framework for automating the bug assignment process. A variety of machine learning algorithms were tried, including Naive Bayes, Decision Trees, Nearest Neighbor. A set of three training approaches were also tried, a retraining method that chose 100 reports at each iteration until the desired accuracy was obtained, a strict method that selected the number of problem reports based on the developer's lifetime contribution and a tolerant method that chose at random problem reports that were proportional to the developer's lifetime contribution. Evaluation of the proposed system was performed on five projects. The overall accuracy results for Firefox was 75% and for Eclipse 70%.

Canfora, et-al investigated bug assignment using text similarity and indexing techniques. A top 1 recall of 50% was reported when using IR based techniques, with a lower performance of 30-50% when using indexing techniques [15, 16].

The impact of the module identifier was investigated by Lin, et-al. Data from a proprietary project, SoftPM, was utilized [19]. The module identifier is a variable that identifies the specific module the bug is associated with. Average accuracy of 77.64% was reported with the identifier is place, and 63% without the use of the identifier.
The use of language was explored in Matter, et-al, when the expertise of the developer was built around the spectrum of vocabulary found in their source code [20]. As new problem reports were submitted, salient information was extracted and compared to the vocabulary and a recommendation on a possible developer was made. Prediction accuracies of 71% were reported on the Eclipse dataset.

Bug tossing graphs, in other words the notion of reassigning a problem to another developer when the original developer cannot fix it, have been explored by Jeong, et-al [21] and Bhattacharya, et-al [22]. In [21] the authors used classifiers and Markov model based tossing graphs to recommend potential developers.

2.5 Summary

Our analysis of the related work shows that at the present time, there is almost no automated approach that can guarantee a significant reduction in the amount of effort that a bug triager performs today. Since there are several time intensive tasks that a triager must perform, ranging from determining if a problem report is new or duplicate, retrieving the primary corresponding to the duplicate, and ascertaining if the report describes the relevant problem or a problem with a different system, it is highly desirable to have a system that alleviates a part of this burden.

The related research also showed us that there is no single “silver bullet” approach that can solve all the problems. Each approach has its limitations and only addresses part of the problem. Moreover, the lack of any experimentation or adequate results when using complete repositories is cause for concern. In order for research to be accepted by industry, we as researchers must demonstrate its feasibility on real world problems.

The major concern with the related work is the employment of techniques without an in-depth understanding of the datasets. Many of these approaches select show results on a small subset of problem reports, without a guarantee on how well these approaches will scale on the entire dataset. In the next chapter, we provide an analysis of the datasets used in this thesis, so as to better understand the characteristics of problem repositories.
3 Analysis of Datasets

Before delving into the methodologies for detecting duplicate and primary reports, it is essential to understand and appreciate the differences between the datasets found in literature. To date, the only evaluation of problem repository characteristics was done by Anvik, et-al in 2005 [23]. The datasets chosen in [23] are the complete Eclipse dataset until August 2005, and a fraction of the Firefox dataset chosen from May 2003 to August 2005. No effort has been made in providing a comprehensive summary of any large-scale datasets and highlighting the key differences and similarities between them from the perspective of the reports and reporters.

The three datasets chosen for the purpose of this research are the complete datasets from Mozilla, Eclipse and Firefox. These three datasets are chosen as they represent large-scale open source projects spanning over 10 years. Moreover, subsets of both datasets have been frequently used in related literature when evaluating the performance of various models.

Mozilla is a free software community, best known for developing the Firefox browser [32]. The Mozilla bug repository consists of problem reports submitted for the various software systems developed by them. The current dataset contains problem reports for Firefox, Firefox Mobile, Thunderbird, SeaMonkey, and Bugzilla. Mozilla consists of a central development chain, called Core, which spawns customer releases. Thus, it is possible that a problem observed in any of the customer releases (Firefox, Firefox Mobile, and Thunderbird) actually manifested itself in the Core system. This is of importance to us as much of prior work ignores such problem reports due to the complexity of detection. By utilizing the entire dataset we are able to ensure that all problem reports are accounted for and no reports are excluded.

Eclipse is a multi language Integrated Development Environment (IDE) developed by the Eclipse Foundation [33]. Unlike Mozilla, Eclipse lacks a central core and all problem reports are associated with the Eclipse project.

Open Office was an open source office suite that was released as a competitor to Microsoft Office [56]. The initial version of the software was released on May 1, 2002 and finally closed in April 2011 due to emergence of platforms such as Google Docs. The Open Office framework is still utilized by Apache for Apache Open Office and The Document Foundation for LibreOffice.

While the datasets used in this research have distinctly different sizes (699,085 for Mozilla, 363,770 for Eclipse, and 124,476 for Open Office) and longevity (5,100
days for Mozilla, 3,700 days for Eclipse and 4,912 days for Open Office) at the time of collection, we can still make observations regarding the similarities and differences between them.

Comparing the three repositories we notice that the biggest difference between them lies in the percentage of DUPLICATE and FIXED reports. In Mozilla, 22% of the reports are DUPLICATES when compared to only 14% in Open Office and 11% in Eclipse. Worse yet, in Mozilla only 35% of the problem reports have been FIXED compared to 40% and 56% in Open Office and Eclipse respectively. In a project such as Mozilla with a diverse user base, many of whom may not be adept at ensuring their reports are not duplicates, one may expect the triage and development team to expend their resources in ensuring duplicate reports are properly classified in addition to fixing actual issues. This notion is echoed in Anvik’s work by some Mozilla developers when they state: “It’s essential that duplicates be marked without developers having to look at them, there are just so many.” [23].

Through the course of this chapter we will answer the following research questions:

- What is the dynamics of software repository evolution over time in terms of the number of reports per year, the number of users and the number of reports per user?
- Does the type of problem reports (DUPLICATE, WONTFIX, WORKSFORME, INVALID, FIXED, OPEN, INCOMPLETE) change as the repository evolves?
- Does the maturity of the user base (measured as the number of novice submitters, infrequent submitters and frequent submitters) evolve over time?
- Does user maturity affect his / her propensity to submit specific types of problem reports (DUPLICATE, WONTFIX, WORKSFORME, INVALID, FIXED, OPEN, INCOMPLETE)?
- Does user maturity change over time as users transition from novice users to experienced users?

### 3.1 Overall User Behavior

Problem reports are submitted by human users; thus, it is essential to also understand how users behave within the system. Eclipse consists of 35,277 distinct users,
as compared to 148,074 in Mozilla and only 25,126 in Open Office. While Mozilla has a highest number of submitters, each submitter in Eclipse submits more reports on average (10.3 for Eclipse vs. 4.72 for Mozilla). The average number of reports per reporter in Open Office is similar to Mozilla with 4.95 reports per submitter. Moreover, only 59% of the submitters in Eclipse submit a single problem report compared to 68% and 71% of the users in Open Office and Mozilla respectively. Thus, the users in Eclipse are far more active than in Mozilla or Open Office. The high level demographics of the users are summarized in Table 4 below.

Duplicate problem reports are an integral part of any problem reporting system. Understanding the overall behavior of duplicate submitters has not been reported in the related literature. 32% of all users in Eclipse, 41% of all users in Mozilla and 33% of all users in Open Office submit duplicate problem reports. Given the more diverse user base in Mozilla, a higher number of users submitting duplicate reports is to be expected. The average number of duplicates per submitter in Mozilla and Open Office are similar, with 2.55 and 2.24 duplicates per submitter. In contrast, duplicate submitters in Eclipse submit, on average, more duplicates at 3.73 reports per submitter. Given that Eclipse is a development environment, such a behavior can be indicative of disgruntled developers submitting reports in an attempt to get their issues resolved. In all three projects, around 2 – 3% of the duplicate reports were from users submitting successive reports describing the same issue.

It is also essential to understand whether duplicate reports are submitted by
distinct users, or the result of one user repeatedly submitting the same issue. 12% in Eclipse, 10% in Open Office and 6% in Mozilla were submitted by the same user who submitted the original primary report. The repeated submissions in Eclipse and Open Office may be due to frustrated developers. In particular for Eclipse, we noted that on average each duplicate submitter is submitting 3.73 duplicates.

One of the biggest challenges when performing research in open source repositories is the presence of inverse duplicates. In the ideal situation, a duplicate report will have a identification number that is larger than the primary report that describes it. However, there are instances when a problem report is submitted that does not provide sufficient information to the developers or is challenging to fix. In such instances, the developer must wait for a “duplicate” report to be submitted that describes the same issue. Thus, the first report, despite having a lower identification number, is marked as a duplicate of the later report. All such duplicate reports are commonly called inverse duplicates and exist within all three datasets. For example, in Open Office problem report 56 is marked as a duplicate of 17198, when in reality report 17198 should have been marked as a duplicate of 56. Failing to correct inverse duplicates by reassigning the status (primary or duplicate) labels will mean that the first report in such a sequence cannot be matched to any similar reports in the past. In Eclipse 11% of all duplicates are inverse duplicates, as compared to 14% and 16% in Mozilla and Open Office respectively.

The high level demographics of the users submitting duplicates are summarized in Table 5.
<table>
<thead>
<tr>
<th></th>
<th>Eclipse</th>
<th>Mozilla</th>
<th>OpenOffice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of duplicate report submitters</td>
<td>11,181</td>
<td>61,205</td>
<td>8,228</td>
</tr>
<tr>
<td>Percentage of all submitters who submit duplicates</td>
<td>32%</td>
<td>41%</td>
<td>33%</td>
</tr>
<tr>
<td>Average number of duplicates per submitter</td>
<td>3.73</td>
<td>2.55</td>
<td>2.24</td>
</tr>
<tr>
<td>Median number of duplicates per submitter</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Maximum number of duplicates by a single submitter</td>
<td>524</td>
<td>640</td>
<td>135</td>
</tr>
<tr>
<td>Number of submitters who submit only a single duplicate</td>
<td>7,408 (66%)</td>
<td>44,685 (73%)</td>
<td>6,089 (73%)</td>
</tr>
<tr>
<td>Percentage of duplicates where the original was submitted by the same user</td>
<td>12%</td>
<td>6%</td>
<td>10%</td>
</tr>
<tr>
<td>Proportion of duplicates that are inverse duplicates</td>
<td>11%</td>
<td>14%</td>
<td>16%</td>
</tr>
</tbody>
</table>

Table 5: Summary Of Duplicate Submission Behavior

### 3.2 Evolution of the Dataset over Time

While the high level overall user behavior illustrates the differences between the datasets, and also the similarities, it is essential to delve into how the three datasets have changed over time. The three datasets represent long term projects spanning over 10 years.
3.2.1 Total Number of Reports Per Year

To start, we explore the total number of problem reports submitted per year in Eclipse, Mozilla and Open Office as illustrated in Figure 3. We note a few points in this figure for clarity. Eclipse was conceived in 2001 and has no reports before then. Our collection for Eclipse stopped at the end of 2011, and thus we do not have any reports for 2012. We collected all Mozilla problem reports from 1998 through the first 3 months of 2012; hence, this final year’s total report number does not represent a typical full year of data for the project. Finally, we collected Open Office data from the project inception in 2000 to 2014. However, we do not report on the last two years in order to maintain uniformity with the other two projects.

By looking at Figure 3 we notice that the rate of submitted reports for Eclipse is relatively stable on a per year basis. From years 2004 to 2011, the annual number of reports were nearly uniform. The number of reports for Mozilla, on the other hand, shows two phases of marked growth. From 1998 to 2002 submission frequency to the Mozilla project grew each year. In 2003 the project saw a dramatic decline in the number of reports submitted for that year. Starting in 2003 the project showed a nearly uniform submission rate until 2007. From 2007 onwards Mozilla has shown a growth pattern that has extended into 2011 and beyond. Open Office shows a steady growth in the number of reports per year until 2005, then from 2005 onwards the project shows fewer problem reports being submitted per year.

One can explain these trends in Mozilla from a business perspective. From 1998 to
2002 Mozilla was still a part of the legacy Netscape system. In 2002 Mozilla released Mozilla 1.0, which can explain the sudden spike in problem reports at the time. Between 2002-2003 the Mozilla project created the Mozilla Foundation and released Firefox 1.0. Updates were made to Firefox on an annual basis with major releases in 2005, 2006 and 2008. In 2008 Firefox 3.0 was released which then moved to a shorter 6 month update cycle until 2011 with Firefox 4.0. Following that, Mozilla went to a rapid release cycle with new major releases every 6 weeks. Thus, the stabilization in problem reports from 2004 to 2008 could be due to the annual releases ensuring clients were using a stable version. The progressive growth from 2009 onwards can be due to the shorter release cycles and rapid release, both of which would entail more releases and potential issues to manage. For Open Office, the emergence of Google Docs in 2006 allowed users to utilize spreadsheet applications without needing to purchase office products. The growth of Google Docs and Office Online resulted in fewer users seeking open source office applications. Open Office was finally shut down in 2011, successor projects using some of the original code include Apache Open Office, LibreOffice and NeoOffice.

3.2.2 Report Types per Year

Eclipse has demonstrated a stable rate of problem reports per year (recall Figure 3), but Mozilla and Open Office are more erratic. We suspect that both Mozilla and Open Office have undergone evolution not only regarding report frequency but also in the types of reports being submitted. Each problem report within the repository can be categorized as OPEN, FIXED, DUPLICATE, WONTFIX, WORKSFORME, EXPIRED, INVALID or INCOMPLETE. Figure 4 shows changes in report types over time represented in an area chart; the areas in the figure correspond to the percentages of each report type at that time.

In the Eclipse dataset, we notice that the percentage of OPEN reports (ones that have not yet been fixed) has grown over time. In contrast INVALID, WORKSFORME and WONTFIX reports have shown a dramatic decrease in numbers over time. One possible explanation is that users have become more cognizant and are ensuring they are not submitting inconsequential issues. The percentage of DUPLICATE reports remains stable over the lifespan of the project. We suspect the increase in the proportion of OPEN problem reports over time could be due to the new reports being more challenging to fix or perhaps having insufficient resources to manage the increasingly complex software systems.
Figure 4: Changes in Report Type Over Time
In contrast, Mozilla shows a marked difference in report types over time. Similar to Eclipse, the percentage of OPEN problem reports have increased over time, while the percentage of WONTFIX reports have stabilized to around 3%. However, from year 2001 to 2006 there was a dramatic change within the repository, which may be primarily caused by the public release of the Firefox browser in 2002. During this time frame, the percentage of DUPLICATE and INVALID reports increased with a corresponding decrease in the percentage of FIXED reports. We anticipate that the development team spent significant time triaging such reports as opposed to fixing actual problems.

On the other hand, the Open Office dataset shows a dramatic increase in the percentage of OPEN reports over time with a corresponding decline in the percentage of FIXED reports. This is caused by the reduction in usage after the year 2005. From 2000 to 2002, there were few users submitting a limited number of reports. As a result most issues were FIXED, with few problems remaining OPEN. However, the increased popularity of Open Office from 2003 to 2005 saw a higher percentage of OPEN problems and a decline in the percentage of FIXED issues. Similarly, this influx of new users meant more the proportion of DUPLICATE reports also increased.

### 3.2.3 Priority and Severity per Year

As users submit problem reports, they must provide a severity value chosen from the following list: blocker, critical, major, minor, normal, trivial and enhancement. These values define how the user perceives the problem report. The most severe, a blocker problem will result in development stopping until it is resolved; on the other hand, an enhancement represents a feature that the user believes is a general system improvement. Once a report has been submitted, the development team assigns a priority to each problem report to define how quickly it will be addressed. A priority value can take any of the following values: P1, P2, P3, P4 and P5. Projects in Mozilla have a priority of BLANK as an additional option, indicating it has not been assigned a priority. The priority value defines the importance assigned by the development team. The principal distinction between severity and priority is that the former is user-assigned while the latter is developer-assigned.

Understanding if the priority or severity of problem reports changes as the software evolves will allow us to determine if the user experience causes them to submit reports of specific types. Identifying high priority or severity reports, and the users
Figure 5: Changes in Priority and Severity Over Time
who are most likely to report them, could be of practical importance to ensure proper resource allocation. In Figure 5 we explore the changes in priority and severity over time.

In the Eclipse project, the P3 priority becomes drastically dominant over the other values. Because the Eclipse project assigns P3 as the default priority, developers may have determined over time that changing the priority of a problem report did not change the time required to assess the final outcome for it; thus, they tend not to change the priority from its default value. The severity values exhibit a relatively uniform trend over time, indicating that even as the project matures the users submit the same ratio of report severities. However, this is a global perspective and does not address the maturity of the individual users.

The Mozilla project uses the “BLANK” priority by default. With time, the “BLANK” priority has become overwhelmingly predominant over the other priority values. In speaking with Mozilla engineers, we were informed that the selection of priority is heavily dependent on the team. One can thus surmise that many development teams have elected to disregard priority value assignment as the added priority information may not provide additional benefits in practice. The Mozilla project, in contrast to Eclipse, exhibits changes in severity over time. The proportion of “normal” severity reports is lowest around 2003 and has more recently been increasing, while “minor”, “major”, and “enhancement” severities have been decreasing.

Open Office, similar to Eclipse, favors the P3 priority as the default priority for all problem reports submitted to the repository. In 2011, the P3 trend showed a distinct change in behavior compared to the past years by showing a 7% decline in overall ratio. In contrast, the P5 (lowest) priority showed a 5 fold increase. Given that the Open Office project was closed in 2011, developers may have simply ignored new reports as they sought out new ventures. Severity values in Open Office have predominantly been trivial, with barely any usage of the values. As the project came to a close in 2011, the percentage of “blocker”, “critical”, “major”, “minor” and “normal” values increased when compared to “trivial”. One reason for this could be the primary user base no longer utilized the product, and the new realm of active users were unaware of the impact of their problems.

### 3.2.4 Changes in Users per Year

Here, we define a “user” as one who has submitted at least one bug report to a corresponding repository. Problem reporting systems serve human users, and through
the lifespan of a project the number of users grows or diminishes. It is essential to understand the effect of users as they become more involved and their impact on report type, severity and priority. Figure 6 illustrates the number of distinct submitters in our three datasets on a per year basis, as well as the average number of reports submitted by them.

The user base in Eclipse changes slowly and plateaus at just over 5,000 distinct submitters. The average number of reports submitted by a distinct user stabilizes at around 7 problem reports per year. However, Mozilla exhibits a dramatic shift in behavior. The number of distinct users skyrockets in 2002, around the release of Firefox, and the average number of reports per user drops dramatically. This indicates a sudden influx of novice users into the system, who manifest through an increase in poor quality reports around this time. OpenOffice sees a peak in the number of distinct users in 2005, followed by a gradual decline to less than 1,000 distinct users from 2011 onwards. The average number of reports per user peaked in 2001 when the project was released, but since then has remained steady through the lifespan of the project.

Also important is understanding how the maturity of users changes over time. We first create four classes based on the number of reports each user has submitted: 1 (the first or only report submitted by the user), 2-10 (the second through 10th reports submitted by the user), 11-50 (the 11th through 50th reports submitted by the user) and 50+ (the 50+th report submitted by the user). This will enable us to determine if the maturity of the users changes, or if the system has a similar proportion of novice and mature users each year. Figure 7 illustrates the changes in

Figure 6: Changes in Distinct User Behavior Over Time
Figure 7: Changes in User Maturity Over Time

maturity over time.
Again, Eclipse demonstrates year to year stability with relatively uniform user maturity. Mozilla shows a clear increase in novice users around the release of Firefox in 2002, which coincides with the increase in unfixed reports seen in Figure 4. Open Office on the other hand shows a progressive increase in the number of novice and first time users and a decline in the number of mature users submitting over 50 reports. From Figures 6 and 7 we can begin to form an understanding of how the repository changes with the influx of new users.

In Mozilla, a moderate negative correlation (−0.66) exists between the percentage of first time and one time submitters and the percentage of FIXED reports. Also, a strong positive correlation (0.79) exists between the percentage of first time and one time submitters and the percentage of INVALID reports. That is, as more inexperienced users enter the system, the percentage of FIXED problem reports tends to decrease while the percentage of INVALID reports tends to increase. From a practical perspective, developers must remain cognizant of the changes in the repository, and allocate resources appropriately to ensure they efficiently manage problem reports. The massive influx of users into the Mozilla repository around 2002 created a dramatic change in the repository, which meant developers had to reallocate resources to manage the increased demands. In Open Office, a strong negative (−0.92) correlation exists between the percentage of first and one time submitters and the percentage of FIXED reports. Also, a strong positive correlation (0.82) exists between the percentage of first and one time submitters and the percentage of OPEN problems. Open Office represents a software in decline, with the crux of the mature users having moved on.

3.3 User Type Analysis

Understanding how the different categories of users submit reports of the aforementioned types will allow us to gain insight into the human aspect of problem reporting systems. Figure 8 summarizes the proportion of users within each dataset and their contribution to the overall size of the repository. We notice in Eclipse only 9% of the users submit more than 10 problem reports, and yet contribute 83% of all problem reports. In Mozilla, 4% of the users submit more than 10 problem reports yet contribute 68% of the dataset. In Open Office, 4% of the users submit more than 10 reports and yet contribute 68% of all problem reports. Thus, despite all three datasets having a predominant set of one time submitters, the vast majority of all problem reports are submitted by moderate and super users.
Figure 8: Proportion of Reports by User
Figure 9 illustrates the changes in the proportion of reports of various types submitted by users of different categories. In Eclipse, 18% of all reports submitted by one time submitters are duplicates, whereas for super users this number drops to 10%. Similarly, in Mozilla 30% of all reports submitted by one time users are duplicates as compared to only 14% by super users. For Open Office, 21% of all reports submitted by one time submitters are duplicates, whereas for super users this number drops to 10%. In contrast, for Eclipse only 44% of the reports submitted by one time users are actual issues (marked as open or fixed) as compared to 73% for super users. A similar trend was found in Mozilla, 17% of reports by one time submitters were actual issues while 68% of reports submitted by super users were actual issues. In Open Office, 36% of all reports submitted by one time submitters described actual issues, while 77% of all issues submitted by super users were actual issues. The above analysis paints a clear picture, user experience weighs heavily on the propensity for a user to submit duplicate reports or actual issues that are relevant to the development team.

Figure 10 illustrates the priority and severity values for the various user types. The trend that is apparent is that one time submitters often perceive their problems to be very important, as noted by more dramatic severity values. In contrast, super users are careful when assigning high severity values and choose to utilize blocker, critical and major values sparingly. However, this trend was not observed in Open Office. The majority severity value is the “trivial” value, and with user maturity this value increases in usage. An overall correlation was not found between the priority and severity for the various user types.

The length of prose submitted for a problem report can also provide insights into behavior of users. In Figure 11 we explore the median length of problem reports submitted by one time, seldom, moderate and super users. We notice that for Eclipse and Mozilla, as users submit more reports the median report length decreases. However, in Open Office the report size remains constant for all user types. For Eclipse and Mozilla, this could be explained by the fact that as users gain more maturity they are likely to use fewer words to describe the problem. On the other hand, for Open Office the user base may not be diverse enough to have significant user type variations.
Figure 9: Type of Reports by User Type
Figure 10: Priority and Severity by User Type
3.4 User Maturity over Time

While the overall behavior of the users provides us with insights as to how users of different maturity behave, it is essential to understand how users themselves evolve over time. All users were at one point first time submitters. Some of these users will, over time, become multi-submission users by continuing to submit to the repository. In this subsection, we investigate the changes in these users’ behavior over time by exploring the evolution of super users (those submitting more than 50 reports). That is, we look at how super users’ reports change as they transition from first time submitters to seldom users to moderate users and finally to super users. Figures 12, 13 and 14 illustrate the behavior of super users who had more than 50 submissions at the time of data collection for Eclipse, Mozilla, and Open Office respectively. They show the relative proportion of report types, severity, and priority as they evolved into super users.

In all three projects we notice that as super users transition from first time submitters to super users, they begin submitting more OPEN problem reports. In Eclipse, the super users tend to submit the same rate of DUPLICATE reports throughout their maturity phase; however, in both Mozilla and Open Office the super users demonstrate a decline in their propensity to submit DUPLICATE problem reports. In Eclipse and Open Office, the super users tend to submit a similar
Figure 12: Evolution of the Super User for Eclipse
Figure 13: Evolution of the Super User for Mozilla
Figure 14: Evolution of the Super User for OpenOffice
portion of FIXED reports, but in Mozilla as super users become more experienced, they tend to submit more FIXED reports. This behavior indicates that, with experience, super users in Mozilla are more likely to submit issues that represent real issues or challenging issues that require extensive effort to resolve. In both projects super users showed a decline in their propensity to submit INVALID and WORKS-FORME reports, thus indicating they are more careful in ensuring their problems are actually associated with the product at hand or represent real problems. In both datasets, the proportion of WONTFIX reports stayed stable as the super users gained experience.

In both Eclipse and Mozilla, transitioning super users began submitting fewer trivial, enhancement and minor problems. In Eclipse, unlike in Mozilla, super users submitted fewer blocker requests as they gained maturity. In Eclipse, major and critical requests stayed the same over time. On the other hand, in Mozilla critical requests stayed the same, but major requests showed a decline over time. Both projects showed a progressive increase in normal reports over time. From the data we understand that as users mature they take more time to determine the impact of their issues by submitting fewer frivolous requests (trivial, enhancement and minor) and ensuring the problems they are submitting represent actual concerns. In Open Office, as users gained experience they chose to submit reports that were categorized as normal as opposed to trivial. The utilization of severity values in Open Office is biased heavily towards trivial, unlike Eclipse of Mozilla.

In all three projects, the default priority (P3 in Eclipse and Open Office and BLANK in Mozilla) was more predominantly used as the users gained maturity.

Figure 15: Evolution of the Super User (Report Length)
As discussed earlier, the development teams may have chosen to simply retain the default priority as opposed to assigning a priority.

As illustrated in Figure 15, the super users in both Eclipse and Mozilla demonstrate a more succinct writing style as they become more experienced by using fewer words to describe issues. This behavior can be attributed to some super users being developers and realizing that using excessive prose can have the detrimental effect of needlessly wasting developer time. However, a similar behavior pattern was not noted in Open Office.

### 3.5 Architectural Analysis

Thus far we explored the many human factors in open source problem repositories. The architecture of problem repositories can affect how easily one can detect problem reports. Table 6 compares the three datasets regarding Product, Component and Classification values [34]. Classification is a top level categorization that is used to filter problem reports into a distinct broader issue area. Product is a second level category below Classification. Once a Product has been chosen the correct Component can be chosen which best describes the area that the problem report is associated with. Despite being a much larger dataset, Mozilla only allows users to select between one of 5 Classifications, 940 Components and 76 Products. On the other hand, Eclipse has 11 Classifications, 822 Components and 178 Products. Similarly, Open Office has 7 Classification values, 153 Component values and only 41 Product values. Thus, a user submitting a report into Eclipse or Open Office is better able to target the correct issue area given the finer granularity in their categorizations. Intuitively, one would expect that associated primary and duplicate reports would share one or more of the three categorizations (Classification, Product and Component). In Eclipse 96% of all duplicate reports share at least one of the three categorizations in common with its primary, in contrast to only 87% in Open Office and 71% in Mozilla. This is primarily due to the architecture of the Mozilla and Open Office system wherein a main Core development chain can spawn product releases such as Firefox, Bugzilla, Thunderbird, etc in Mozilla and Writer, Calc, Impress, Draw, etc in Open Office. Thus, it is likely that an issue found and marked as an issue in Firefox may actually be manifesting from a bug in the Core codebase.

The architecture of the repository plays a critical role in our ability to detect specific types of reports. For example, one notices from Table 6 that primary and duplicate reports in Eclipse share all three categorizations in common for 74% of all
duplicate reports. Thus, it is possible that the categorization values can be used to improve the search procedure for duplicate.

### 3.6 Duplicate Group Sizes

Understanding the size of each group of similar reports is also essential. For example, large groups of reports can have very diverse language, which can cause issues in frequency based methods. In Eclipse 82% of all problem reports are singular primaries (that is, they have no associated duplicates). In contrast, in Mozilla only 69% and in OpenOffice only 77% of all reports are singular primaries. The maximum number of duplicates associated with each group of duplicates is higher for Mozilla (279 for Mozilla vs. 98 and 50 for OpenOffice and Eclipse respectively). In Eclipse, 99% of all problem reports belong either by themselves (singular primary) or in a group consisting of at most 9 other reports. In comparison, 99% of all reports in Mozilla either belong by themselves or in a group with at most 58 other reports. Finally, in OpenOffice 99% of all problem reports either belong by themselves or in a group of at most 23 other reports. Thus, while both projects may have the same median group size, Eclipse has smaller and fewer groups of reports when compared to Mozilla and OpenOffice. This effect is illustrated in Figure 16; we note that the majority of the Eclipse dataset is encompassed by small groups. The vertical axis represents
the cumulative proportion of the dataset, while the horizontal represents the group size (primary + duplicate). These prior set of differences may be due to the nature of the two projects. Eclipse is a development environment, thus one might expect diligent users who are less likely to submit duplicate reports without spending effort in ensuring the problem has not been reported before.

![Graph showing group size comparison](image)

**Figure 16: Group Size Comparison (Mozilla and Eclipse)**

### 3.7 Discussion

Throughout this section we explored the many differences, as well as similarities, between Eclipse, Mozilla and Open Office. We summarize the main findings as a series of observations and suggestions.

1. **Observation:** all three datasets have a large number of inverse duplicates, wherein the primary has a higher (submitted later in time) identifier than the duplicate. See Table 5.

   **Suggestion:** in order to maintain consistency, these inverse duplicates must be fixed periodically so that the primary report always has the lowest report number in a duplicate group. Failure to do so will result in automated systems missing duplicate matches.

2. **Observation:** priority values in all systems are often left at the default value, limiting their meaningfulness. See Figure 5.
Suggestion: priority values should not be included for predicting problem reports.

3. Observation: historically, duplicate classification in Eclipse has been easier due to the more structured user base who consist mostly of developers due to the nature of the software [8], [11].
   Suggestion: no silver bullet approach can be applied, methodologies must utilize the specific characteristics of each dataset.

4. Observation: a small number of users submit a vast majority of the reports. See Figure 8.
   Suggestion: the 80-20 rule, wherein 80% of all reports are submitted by 20% of the users, suggests that more attention must be paid to regular users as they submit many reports that are of better quality and describe real issues.

5. Observation: heavy users of the system submit much higher quality reports. See Figure 9.
   Suggestion: when assigning reports to developers to fix, priority must be given to reports submitted by frequent users.

6. Observation: one time submitters often overestimate the severity of their issues. See Figure 10.
   Suggestion: if a reporter is noted as a first time submitter, disabling higher severity (blocker or critical) values will ensure developers are not given issues of exaggerated importance.

7. Observation: the release of a major consumer product, such as Firefox, causes a large influx of users that creates more low quality and duplicate reports in the dataset. See Figures 6 and 7.
   Suggestion: allocating more triaging resources immediately after release can prevent a backlog of untriaged reports.

8. Observation: novice users typically submit more text than regular users. This may require developers to devote more time in comprehending the text. See Figure 11.
   Suggestion: novice users can be provided with more user-selectable menus that describe the problem in addition to the free text that they submit.
9. Observation: Eclipse has a systematic architecture which results in 74% of all duplicates having Product, Component and Classification values in common with their corresponding primary. See Table 6.

Suggestion: for systems in which Product, Component and Classification values are easy to identify by users, use Product, Component and Classification values to further narrow the search space for duplicates.
4 The Automated Framework for Classifying Problem Reports as Primary or Duplicate

Much of the related literature focuses on matching a given (known from the ground truth) duplicate report(s) with its corresponding primary report. Related research answers the question: “Given that a report is a duplicate, what are its most likely matches?” In practice, a priori knowledge that the new report is, indeed, a duplicate is a strong assumption. In reality, prior to determining the potential matches for a duplicate, one must first ascertain whether the report is indeed a duplicate. While one may simply assume that reports that do not match well any existing ones should be primaries, we show that setting up the thresholds on “a good match” is far from trivial.

In this chapter we develop a tool to automatically determine whether an incoming report is primary or duplicate. Our framework relies on utilizing three sets of information: how well does the report match to prior reports, what are the specific characteristics of the report, and what are the characteristics of the person submitting the report. The first leverages the notion that a pool of similarity measures can allow us to closely determine how well a report matches to past reports. The second allows us to find report specific information such as length of prose, time list last primary/duplicate, report priority/ severity, etc. The third allows us to utilize the reporter credibility based on the notion that good users submit good reports.

Our contributions are as follows:

- A novel framework for automatically classifying problem reports as primary or duplicate by leveraging similarity, report and user attributes
- Demonstrate that training set size does not impact the ability to classify a report as primary or duplicate
- Demonstrate that the project type has an impact on the ability to classify a report as primary or duplicate

4.1 Dataset Definition

For the purpose of this research we utilized three datasets: Eclipse, Open Office and Firefox. The dynamics of the three datasets, in terms of the number of primary and duplicate reports, is defined in Table 7.
### Table 7: Dataset Definition

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Primary Reports</th>
<th>Duplicate Reports</th>
<th>Primary Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>321,972</td>
<td>41,748</td>
<td>88%</td>
</tr>
<tr>
<td>Firefox</td>
<td>86,120</td>
<td>25,085</td>
<td>77%</td>
</tr>
<tr>
<td>Open Office</td>
<td>104,634</td>
<td>19,842</td>
<td>84%</td>
</tr>
</tbody>
</table>

#### 4.2 Document Preprocessing

In order to create a dataset suitable for experimentation we performed order normalization, tokenization and stemming, and created a custom stop word list. These steps are standard procedures, and will also be utilized when classifying duplicates.

In ideal situations when reports in the repository are ordered chronologically in time, a primary problem report must have an identifying numeric value that is lower than all its duplicates. Moreover, a duplicate report should never be marked as the primary of another problem report. However, the data set contents indicate that these rules are not always followed in practice. If a user submits a report that does not have sufficient information to fix the issue, the development team may decide to wait for a later report and develop a fix at that point in time. Thus, this later report will be assigned a primary label while the earlier report is marked as a duplicate. Such ground truth markings in the data set must be corrected to recreate the submission order. To mitigate time-related reordering problems, we swapped the identification numbers of such pairs. Occasionally, a report may be assigned a primary status while it is, in fact, a duplicate of an existing report. Such issues arise when the triage team incorrectly assigns the status, thereby creating a split in the repository.

Tokenization is the process by which each problem report is reduced to a simpler form by removing punctuation marks and converting all uppercase characters to lowercase characters. The diversity of the English language necessitates the use of stemming methods that reduce each word to its root form. The approach utilized in this thesis is the classic Porter’s stemming algorithm implemented in Perl [47].

Stop words are unnecessary words that add little to the semantics of each problem report. For example, ‘the’ is a common word, but does not help in classifying problem reports. There exists no standard stop word list. Furthermore, the contents of the stop word list are context and domain dependent. We utilize the list of 114 stop words, to exclude the most common words of the English language.
4.3 Time and Space Constraints

The time and space constraints for this research involving open source repositories have not been discussed in prior literature. The number of similarity computations needed for each dataset can be computed by finding the sum of the divergent series 
\[1 + 2 + 3 + 4 + \ldots + n,\]
where \(n\) represents the total number of problem reports. Mathematically, this sum can be expressed as: 
\[
\frac{n(n+1)}{2},
\]
where \(n\) is the number of reports, \(a_1\) is the value of the first report, and \(a_n\) is the value of the last report. Or, it can be approximated to \(\frac{n^2}{2}\), where \(n\) is the number of problem reports. Or, in general we can consider the problem to be \(O(n^2)\).

For Eclipse the total number of similarity scores is \(6.6 \times 10^{10}\), for Open Office \(7.7 \times 10^9\) and for Firefox \(6.2 \times 10^9\). Each similarity score contains 100 bytes of information. Thus, for all three projects combined one must store \(8 \times 10^{12}\) bytes of data for each similarity method. Considering the approach defined herein utilizes 4 base methods, a conservative estimate of \(3.2 \times 10^{13}\) bytes (or 29 TB) of storage is needed.

Processing of the datasets were conducted at the West Virginia University Department of Statistics computer labs using a cluster of 25 Apple iMacs. Each iMac contained a 250GB Solid State Drive (SSD), with a 2.8 GHz Intel i7 Quad Core processor with 8GB of RAM running OS X. Given university and storage constraints, processing was only possible during weekends at an average rate of 500 files per machine per day. Thus, processing all three datasets took approximately 40 weeks of processing due to data loss caused by file deletion or process termination. The Amazon Elastic Computing Cloud (EC2) offers adequate computational resources in a high performance cloud environment [59]. Conducting our experiment on EC2 would require the “Storage Optimized - Current Generation i2.8xlarge” system containing 104 ECUs, 244GB of RAM and 6.4TB of SSD storage. At a cost of $6.820 per hour, we estimate the total cost to process the all three data set would be just under $10,000 when factoring in the EC2 cost as well as the cost to purchase 29TB of local storage.

Given these constraints, one assumption had to be made for the larger dataset (Eclipse) which was relaxed for the Firefox and Open Office. The assumption made for Eclipse is to assume that an oracle exists that can define the duplicate groups for all reports in the past. The decision to relax the assumption for Firefox and Open Office is to simulate an environment that is as close to the real world as possible where an oracle may not exist who can define each duplicate group.
4.4 Similarity Measures

We created a suite of 24 text similarity measures using combinations of three simple techniques: base text similarity methods, time windows, and document factors. The two base methods are the cosine similarity and the longest common subsequences. In our experimental setup, the two base methods are used to compare the Title or the Title and the Summary fields of new reports to every prior report in the repository. Time windows and document factors are used to narrow the otherwise huge search space. Each of these techniques create parts of the similarity measure, as described below.

For the Eclipse dataset, we applied a word weighting scheme based on the group centroid approach. Each word is assigned the weight as follows:

\[
\text{WordWeight}_i = \frac{\text{Freq}(Word_i)}{\text{Num}(Documents)}
\]

where the numerator is the number of times the \(i^{th}\) word occurs in the cluster of similar problem reports and the denominator is the number of problem reports in the cluster. In [7] we demonstrated the effectiveness of the group centroid approach over TF*IDF as a word weighting scheme.

Similarity between two documents was measured using cosine similarity and longest common subsequences. Cosine similarity is a vector space based measure for assessing similarity between two problem reports:

\[
\text{Cos}_{(A,B)} = \frac{A \cdot B}{\|A\| \|B\|}
\]

where \(A\) and \(B\) represent two reports in the repository, \(A \cdot B\) represents the dot product of the words similar to the two reports, and \(\|A\|\) and \(\|B\|\) represent the respective sizes of considered problem reports. The Longest Common Subsequence algorithm evaluates document similarity based on the order of words. The longest common subsequence (LCS) is the longest common substring in the two strings. LCS has been widely applied, with uses ranging from molecular biology to the Unix Diff command [46, 47]. The LCS between the problem report \(X\) of size \(M\) and report \(Y\) of size \(N\) is computed following the Algorithm 1.
Algorithm 1 The Length of Longest Common Subsequence

X[1..m] is a string and Y[1..n] is a string
C[m,n] is the length of the LCS of X and Y

function LengthLCS(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
        C[i,0] = 0
    for j := 0..n
        C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]

4.5 Search Space Optimization

As the number of problem reports in the repository increases over time, determining the similarities of an incoming report with all the existing ones by analyzing all prior reports becomes a long process. Moreover, the likelihood of identifying a false primary match is likely to increase with the size of the repository. In order to offset this problem, a time window is used to compare the new report only to the N most recently submitted reports. For the purpose of our research, for the Firefox and Open Office dataset we utilize two different time windows of 1,000 and 2,000 nearest reports along with all prior reports. In contrast, for the Eclipse dataset we compare to the 1,000, 2,000 and 70,000 nearest groups. The smaller sizes of 1,000 and 2,000 are chosen to capture reports in close proximity. In Chapter 5 we demonstrate that duplicate reports are submitted in quick succession. The size of 70,000 in Eclipse is chosen as 95% of duplicates appear in the 70,000 nearest groups.

Product, Component, and Classification fields in problem reports allow developers to pinpoint the exact location of the bug [34]. ‘Classification’ represents top level categorizations used to filter problem reports. Thus, one may expect that a duplicate report shares one or more values in common with its corresponding primary, this particular facet of problem repositories was demonstrated in Section 3.5. In order to utilize these values, we multiply the base similarity score based on how many of the architectural values match. If no architectural values match, then the
score is left as is. If only one architectural value matches, then the score is multiplied by 2. If all architectural values match, then the score is multiplied by 4. Through the course of this thesis, we call the architectural match weighted scores the "factor" weighted scores.

4.6 Generating Similarity Measures

We generate the 24 similarity measures between problem report pairs as follows.

- Two instances of each of the two base methods. One instance uses only the report’s Titles, while the other uses the Titles and Summaries concatenated together, thus creating the 4 similarity score variants.

- Three time windows: 1,000, 2,000, and 70,000 nearest groups in Eclipse, and 1,000, 2,000, nearest and all prior reports for Firefox and Open Office.

- The two variants come from the inclusion or exclusion of factors in matching decisions.

Combining the above, we obtain $4 \times 3 \times 2 = 24$ similarity measures. Each similarity measure provides a related but different insight into the similarity between two reports. Given the large number of match scores generated by each problem report for each measure, we summarize the match scores using the top 20 highest values. We order the match scores in each similarity measure for every incoming problem report and consider the top 20 matches, the number commonly used in literature. In principle, reports outside the top 20 list should be of little interest although, due to many sources of imprecision and uncertainty, this is just an assumption which simplifies the analysis and the computation of results.

4.7 Feature Set Description

As new problem reports are submitted, an analyst needs to ascertain their status as primary or duplicate. Our goal is to automate this task. We train a classifier using a small subset of problem reports labeled as primary of duplicates by triagers. Each report is represented as a feature vector that describes the following characteristics:

- The report itself
- The submitter
• How the report matches to all previous reports

When a new report is submitted, our tools create the feature vector in the same format used to represent the reports included in the training set. This test vector, which corresponds to any incoming report, is submitted to the classifier for the prediction of its label (primary/duplicate).

The part of feature vectors which describe the new report’s similarity to all previous ones is generated by summarizing the lists of top 20 matches from each of the 24 similarity measures described above. We must reduce each top 20 list into a set of statistics that describe the list. Intuitively, we want these statistics to easily differentiate primary and duplicate reports. The reason for using descriptive statistics is that the top 20 lists obtained from the 24 similarity measures cannot be used directly to train or test a machine learner.

The feature vectors are described in Tables 8, 9, 10 and 11. The weighted scores in Tables 9 are created by dividing each match score by the square root of the distance between the matching report and the test report. This is performed to add importance to strong matches with reports found in the recent past.

Each report vector consists of 405 attributes associated with a single class label (Status - Primary or Duplicate). These attributes describe a report’s top 20 lists (Table 9), the report itself (Table 10), and the characteristics of the user who submits the report with respect to his or hers prior activity in the same project (Table 11).

The chosen features do not represent the entire corpus of possible features. Given the complexity of the datasets, applying data mining techniques to extract all salient features is left as future work.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value Range</th>
<th>Num. of Vals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>0-1</td>
<td>24</td>
<td>Structure: 24 values based on 24 similarity measures. Range: 0 to 1. The maximum value is used as this represents the highest match score for the given report for the given method.</td>
</tr>
<tr>
<td>Mean</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The mean value is used as this represents the mean match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Median</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The median value is used as this represents the median match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Skew</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The skew value is used as this represents the difference between the median and mean match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Range</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: 0 to 1. The range describes the difference between the best match score in the top 20 list and the worst match score in the top 20 list.</td>
</tr>
<tr>
<td>Jump</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: 0 to 1. The jump describes the largest difference between two consecutive match scores in the ordered top 20 list.</td>
</tr>
<tr>
<td>Jump_loc</td>
<td>Integers 1 to 20</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: Integers between 1 and 20. The jump location describes the position in the top 20 list where the maximum difference between consecutive match scores occurs.</td>
</tr>
<tr>
<td>Reps_to_nearest</td>
<td>Integer</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: Integer greater than or equal to 0. The reports to nearest describes distance to the report in the top 20 list that is closest to the current report.</td>
</tr>
</tbody>
</table>

Table 8: Description of the Feature Vector Representing Non-weighted Match Score Statistics
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value Range</th>
<th>Num. of Vals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The maximum value is used as this represents the highest match score for the given report for the given method.</td>
</tr>
<tr>
<td>Mean_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The mean value is used as this represents the mean match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Median_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The median value is used as this represents the median match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Skew_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on 24 similarity measures. Range: 0 to 1. The skew value is used as this represents the difference between the median and mean match score for the given report for the given method within the top 20 list.</td>
</tr>
<tr>
<td>Range_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: 0 to 1. The range describes the difference between the best match score in the top 20 list and the worst match score in the top 20 list.</td>
</tr>
<tr>
<td>Jump_weighted</td>
<td>0-1</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: 0 to 1. The jump describes the largest difference between two consecutive match scores in the ordered top 20 list.</td>
</tr>
<tr>
<td>Jump_loc_weighted</td>
<td>Integers 1 to 20</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: Integers between 1 and 20. The jump location describes the position in the top 20 list where the maximum difference between consecutive match scores occurs.</td>
</tr>
<tr>
<td>Reps_to_nearest_weighted</td>
<td>Integer</td>
<td>24</td>
<td>24 values based on the 24 similarity measures. Range: Integer greater than or equal to 0. The reports to nearest describes distance to the report in the top 20 list that is closest to the current report.</td>
</tr>
</tbody>
</table>

Table 9: Description of the Feature Vector Representing Weighted Match Score Statistics
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value Range</th>
<th>Num. of Vals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title_size</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than or equal to 1. The title size defines the number of words found in the title.</td>
</tr>
<tr>
<td>Sum_size</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than or equal to 0. The summary size defines the number of words found in the summary.</td>
</tr>
<tr>
<td>Reps_since_last_dup</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than or equal to 0. The reports since last duplicate defines the number of reports submitted since a duplicate report was submitted.</td>
</tr>
<tr>
<td>Reps_since_last_prim</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than or equal to 0. The reports since last primary defines the number of reports submitted since a primary report was submitted.</td>
</tr>
<tr>
<td>Version</td>
<td>Nominal</td>
<td>1</td>
<td>consists of 1 value. Range: consists of textual entries that define the version of the software.</td>
</tr>
<tr>
<td>Operating_system</td>
<td>Nominal</td>
<td>1</td>
<td>consists of 1 value. Range: consists of textual entries that define the operating system of the system where the bug was discovered.</td>
</tr>
<tr>
<td>Severity</td>
<td>Nominal</td>
<td>1</td>
<td>consists of 1 value. Range: consists of textual entries that define the severity of the bug.</td>
</tr>
<tr>
<td>Priority</td>
<td>Nominal</td>
<td>1</td>
<td>consists of 1 value. Range: consists of textual entries that define how soon the bug needs to be fixed.</td>
</tr>
<tr>
<td>Platform</td>
<td>Nominal</td>
<td>1</td>
<td>consists of 1 value. Range: consists of textual entries that define the types of systems the bug can potentially affect.</td>
</tr>
<tr>
<td>Avg_Time_12</td>
<td>Integer</td>
<td>-1</td>
<td>consists of 1 value. Range: Integer greater than 0 or equal to -1. The average time between submission and first comment defines how quickly the development team responds to a user’s submission. A value of -1 indicates that the user has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_comm_by_user</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than 0 or equal to -1. The prior comments by user defines how many comments the user has submitted in the past.</td>
</tr>
<tr>
<td>Prior_comm_on_user_rep</td>
<td>Integer</td>
<td>1</td>
<td>consists of 1 value. Range: Integer greater than 0 or equal to -1. The prior comments on user report defines how many comments have been made in the past on the user’s report.</td>
</tr>
</tbody>
</table>

Table 10: Description of the Feature Vector Representing the Content of Each Report
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value Range</th>
<th>Num. of Vals</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prior_duplicate</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that were duplicate. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_open</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that are still not fixed. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_fixed</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that have been fixed. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_invalid</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that were marked as invalid. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_incomplete</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that were marked as incomplete. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_worksfomre</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that were marked as worksforme. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_wontfix</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that will not be fixed. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_moved</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that were moved. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
<tr>
<td>Prior_expired</td>
<td>0-1</td>
<td>1</td>
<td>consists of 1 value. Range: 0 to 1. It defines the proportion of reports submitted by the reporter prior to the current report that have expired. A value of -1 is used if the reporter has not submitted a report in the past.</td>
</tr>
</tbody>
</table>

Table 11: Description of the Feature Vector Representing Submitter Information
4.8 Classification Procedure

For classification we used the Weka machine learning toolkit [60]. Several learning algorithms were applied in our initial experiments and random forest appeared to be giving us the best and most consistent results [36]. Random forest is an ensemble learner that combines the results from multiple decision trees to form a classifier. The random forest classifier relies on two tunable parameters, namely the number of trees in the forest and the number of random features chosen for each tree. The forest error rate, that is the performance, depends on the correlation between any two trees and the strength of each individual tree. A highly correlated pair of trees, and a tree with a high error rate will increase the error rate. The number of random features chosen affects both the correlation and the strength of the individual tree. Choosing a value that is too low will reduce the correlation but also the strength, choosing a value too high increases both. By default, a majority voting scheme is used. That is, if more than 50% of the trees agree on a class then the sample is classified to that class. By changing the voting threshold, we can create ROC curves and determine performance at various points of interest [61].

Random forest was trained using a set of labeled feature vectors, while the trained classifier predicted whether an unlabeled bug report is a primary or duplicate. We used random forest classifier with 500 trees, as suggested in the original literature, and using $\sqrt{\text{NumberOfFeatures}}$, $2 \times \sqrt{\text{NumberOfFeatures}}$ and $4 \times \sqrt{\text{NumberOfFeatures}}$ random features as opposed to all features. The square root of the total number of features is utilized as it is often selected as the number of random features to choose at each node in a random forest classifier.

Two different representations of the reports from each repository were used: cleaned data and raw data. In cleaned data representation we excluded all problem reports that were marked as INVALID, INCOMPLETE, WORKSFORME, WONTFIX, EXPIRED, and MOVED. The justification for this is the practice of individuals performing the triage. If a problem report is deemed to be of "poor" quality, the triager may not spend any time determining if the report needs to be classified as a duplicate as the problems will not be addressed. Each data set contains numerous examples of poor quality reports which were not given a duplicate label but should have been classified as duplicates. It is impossible for us to correct the labels of all problem reports. Therefore, excluding them from the training and testing data sets will mitigate potential problems with excessive noise in the training and test data.
To the contrary, in the raw data representation, we retained all problem reports. The raw data also represents the most realistic scenario and contains no assumptions on the quality of the reports and the inherent noise in the labels assigned by the analysts. Comparing the prediction results using cleaned versus raw data set representations will provide insights into our method’s robustness and the extent of noise in the ground truth. The proportion of reports in the cleaned and raw datasets are summarized in Table 12. We carefully note that the cleaned dataset in Eclipse and Open Office maintains the natural split of primary and duplicate reports. In Firefox, the cleaned dataset provides a better balance between the two classes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Raw Primary</th>
<th>Raw Duplicate</th>
<th>Clean Primary</th>
<th>Clean Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>321,972</td>
<td>41,748</td>
<td>246,660</td>
<td>36,097</td>
</tr>
<tr>
<td>Firefox</td>
<td>86,120</td>
<td>25,085</td>
<td>33,668</td>
<td>18,468</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>104,634</td>
<td>19,842</td>
<td>70,865</td>
<td>16,723</td>
</tr>
</tbody>
</table>

Table 12: Proportion of Reports in Raw and Cleaned Datasets

We developed the classifier using the retraining method. The classifier is periodically retrained by a set of recent reports throughout the lifespan of the repository. The classifier is trained with, for instance, 10,000 recent correctly labeled reports, and used for predicting the status of the next 10,000 reports as they enter the system. This retraining approach requires triagers to actively correct mistakes made by the classifier to increase prediction accuracy. We analyze the effects of the window size on prediction performance and present results of cyclical retraining as 2,500 (Firefox and Open Office only), 5,000, 10,000, 15,000 (Eclipse only), or 20,000 (Eclipse only) new reports arrive.

The complete list of experiments performed on each dataset is summarized in Figure 17.
Figure 17: Primary vs. Duplicate Experiments - Eclipse, Firefox and Open Office

4.9 Results

We present experimental outcomes by defining a primary report as a positive, and use the following measures to evaluate our performance.

1. True Positive Rate (TPR) - ratio between correctly classified primaries over the total number of primaries.

2. True Negative Rate (TNR) - ratio between correctly classified duplicates over the total number of duplicates. The TNR can also be computed as $1 - FPR$.

3. False Negative Rate (FNR) - the ratio between the number of primaries classified as a duplicate divided by the total number of primaries. The FNR can
also be computed as $1 - TPR$.

4. False Positive Rate (FPR) - the ratio between the number of duplicates classified as a primary divided by the total number of duplicates.

5. ROC Curve (Receiver Operating Characteristic curve) - is a parametric curve that plots TPR versus FPR across a range of random forest (or other classifier) thresholds. In our case, the ROC curve is plotted by changing the default voting scheme from 0.5 to a range of values from 0.05 to 0.95 in increments of 0.05.

The results are presented as answers to the following questions:

1. Does the training set size affect performance?
2. Does the quality of reports affect performance?
3. Does the dataset affect performance?

Statistical analysis of the generated ROC curves are performed using the pROC package found in R [62]. In particular, the DeLong test is applied to test the differences between the area under the curve (AUC) of each method [63]. A two sided test is utilized to test if the differences in the AUC are statistically significant.

Figures 18, 19 and 20 show the results for the Eclipse, Firefox and Open Office projects respectively using the raw and clean data.

Figure 18: Eclipse Dataset with 500 Trees and 12 Random Features Per Tree
4.9.1 Does the Training Set Size Affect Performance?

Hypothesis Test:
NULL: the AUC for larger training sizes is equal to the AUC for smaller training sizes
Alternate: the AUC for larger training sizes is not equal to the AUC for smaller training sizes
The NULL hypothesis is rejected in all three projects (p<0.05) when making pairwise comparisons between the smallest training size to the larger training sizes. Increasing the training set size increases the AUC of the ROC curve. However, the gained area is less than 0.04 for all three projects. This is better exemplified in Figures 18, 19 and 20 where we note that in all three projects, increasing the number of samples in the training set does not practically affect the overall performance.

Thus, we contend that while training set size has a statistically significant impact, the practical differences in the AUC are marginal. A larger training set will require longer time to train. The system, as designed, can be deployed after 2,500 reports in Firefox and Eclipse (or approximately 3 months) and 5,000 reports in Eclipse (or approximately 2 months).

4.9.2 Does the Quality of Reports Performance?

Hypothesis Test:

| NULL: the AUC for clean data is equal to the AUC for raw data |
| Alternate: the AUC for clean data is not equal to the AUC for raw data |

The NULL hypothesis is strongly rejected in all three projects (p<0.05) when making pairwise comparisons between the clean and raw datasets. Using clean data increases the AUC of the ROC curve. However, comparing Figures 18, 19 and 20 for raw vs. clean data, we note that there is no practically measurable improvement in performance in Eclipse. The maximal difference in the AUC for Eclipse was 0.04. In contrast, for both Firefox and Open Office the ROC curve shifts to the top left, indicating a higher TPR and lower FPR. The difference in the AUC for Firefox and Open Office was in the order of 0.1 and 0.06 respectively, that is the clean dataset increases the performance by 10% in Firefox and 6% in Open Office.

This behavior can be explained by the type of users present in each dataset. The developer oriented nature of Eclipse, by virtue of its application, means users are less likely to submit frivolous reports. In contrast, the presence of average users in Firefox and Open Office means they are likely to submit poor quality reports that do not get appropriately triaged. Thus, we contend that report quality is not a sole contributor to performance and is also affected by the project type and quality of users.
4.9.3 Does the Dataset Affect Performance?

<table>
<thead>
<tr>
<th>Hypothesis Test:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL: the AUC for Eclipse is equal to the AUC for Firefox</td>
</tr>
<tr>
<td>Alternate: the AUC for Eclipse is not equal to the AUC for Firefox</td>
</tr>
<tr>
<td>NULL: the AUC for Eclipse is equal to the AUC for Open Office</td>
</tr>
<tr>
<td>Alternate: the AUC for Eclipse is not equal to the AUC for Open Office</td>
</tr>
<tr>
<td>NULL: the AUC for Firefox is equal to the AUC for Open Office</td>
</tr>
<tr>
<td>Alternate: the AUC for Firefox is not equal to the AUC for Open Office</td>
</tr>
</tbody>
</table>

The NULL hypothesis between raw Eclipse and raw Firefox is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Eclipse raw dataset. The difference in AUC between raw Eclipse and raw Firefox was 0.071.

The NULL hypothesis between raw Eclipse and raw Open Office is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Eclipse raw dataset. The difference in AUC between raw Eclipse and raw Open Office was 0.095.

The NULL hypothesis between raw Firefox and raw Open Office is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Firefox raw dataset. However, the difference in AUC between raw Firefox and raw Open Office was only 0.024.

The NULL hypothesis between clean Eclipse and clean Firefox is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Firefox clean dataset. However, the difference in AUC between clean Eclipse and clean Firefox was only 0.019.

The NULL hypothesis between clean Eclipse and clean Open Office is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Eclipse clean dataset. However, the difference in AUC between clean Eclipse and clean Open Office was only 0.033.

The NULL hypothesis between clean Firefox and clean Open Office is strongly rejected with $p \leq 0.05$, indicating that our methodologies work best on the Firefox clean dataset. However, the difference in AUC between clean Firefox and clean Open Office was only 0.052.

We summarize the results from Figures 18, 19 and 20 in Table 13 by presenting the results when recall in both classes are balanced, and when we filter out as many duplicates as possible without compromising on primary recall. The former system acts as a triager aide, the latter ensures 95% of actual problems are sent to the
developer while filtering as many duplicates as possible.

<table>
<thead>
<tr>
<th>Project</th>
<th>TPR</th>
<th>TNR</th>
<th>TNR at 95% TPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>75%</td>
<td>64%</td>
<td>25%</td>
</tr>
<tr>
<td>Eclipse Clean</td>
<td>73%</td>
<td>68%</td>
<td>25%</td>
</tr>
<tr>
<td>Firefox</td>
<td>68%</td>
<td>61%</td>
<td>20%</td>
</tr>
<tr>
<td>Firefox Clean</td>
<td>78%</td>
<td>67%</td>
<td>34%</td>
</tr>
<tr>
<td>Open Office</td>
<td>75%</td>
<td>47%</td>
<td>14%</td>
</tr>
<tr>
<td>Open Office Clean</td>
<td>65%</td>
<td>68%</td>
<td>23%</td>
</tr>
</tbody>
</table>

Table 13: Summary Results for Primary vs. Duplicate Detection

We notice that for all three projects, we are able to correctly recall 68% of the duplicate reports when using the clean dataset. The best overall results are in the Firefox clean dataset, where we achieve 78% recall for primaries, 67% for duplicates and when using it as a filtering scheme we are able to filter out 34% of the duplicate reports. The most stable dataset is Eclipse, with clean and raw datasets offering little in difference. The most problematic dataset is Open Office. Given the demise of Open Office, it is expected that reports within the dataset are incorrectly or simply have not been triaged.

Thus, we contend that the dataset plays an important role in our ability to effectively classify problem reports. Well formed projects, such as Eclipse, are easy to classify, while projects that have failed, such as Open Office, are challenging.

4.10 Result Comparison

4.10.1 Comparison to a Naïve Approach

Intuitively, it would seem that the primary vs. duplicate classification problem can be solved by setting thresholds for the similarity scores. One may expect primary reports to value values below a certain threshold, and duplicates to have values higher than the threshold. One may set up the threshold and if the top match score exceeds the threshold value we can consider the problem report to be a duplicate, otherwise it can be classified as a primary. Since similarity scores can range from 0 to 1, we can generate an ROC curve by evaluating the performance based on TPR and FPR at each threshold. Figures 21, 22 and 23 depict the ROC curves using the cosine similarity measure with group centroid word weighting, and longest common subsequences.
Figure 21: Eclipse - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences

Figure 22: Firefox - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences
<table>
<thead>
<tr>
<th>Dataset</th>
<th>TPR Automated</th>
<th>TNR Automated</th>
<th>TPR Naive</th>
<th>TNR Naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>75%</td>
<td>64%</td>
<td>60%</td>
<td>50%</td>
</tr>
<tr>
<td>Firefox</td>
<td>68%</td>
<td>61%</td>
<td>62%</td>
<td>50%</td>
</tr>
<tr>
<td>Open Office</td>
<td>75%</td>
<td>47%</td>
<td>69%</td>
<td>45%</td>
</tr>
</tbody>
</table>

Table 14: Automated Triaging Framework Compared To Naive Threshold Approach

![Figure 23: Open Office - Naive Threshold Approach with Cosine Similarity and Longest Common Subsequences](image)

**Hypothesis Test:**

NULL: the AUC for the naive cosine similarity approach is equal to the AUC for the automated triaging approach

Alternate: the AUC for the naive cosine similarity approach is not equal to the AUC for the automated triaging approach

NULL: the AUC for the naive longest common subsequence approach is equal to the AUC for the automated triaging approach

Alternate: the AUC for the naive longest common subsequence approach is not equal to the AUC for the automated triaging approach

We strongly reject both NULL hypotheses (p<0.05) for all three projects, indicating that the automated triaging approach is significantly better than the naive approach. We notice both naive approaches are marginally better than a random guess in Eclipse and Firefox, i.e., they do not work for distinguishing primary reports.
from duplicates. For Eclipse, the naive approach misclassifies The improvements of the automated framework over the naive approach is summarized in Table 14.

### 4.10.2 Existing State of the Art

Comparisons can also be made with the work of Feng et al. [55], who utilized 534 randomly sampled reports out of 22,486 possible bug reports (2.5% of the MeeGo data) and achieved 75% for primaries. However, their use of 10-fold cross validation is unrealistic as it assumes training with reports submitted in the future. Our results are significantly better than Hiew’s [1], and the comparison is meaningful since both experiments use the same data.

### 4.11 Time saved by automation

We calculated the average time for a problem report to be assigned a “DUPLICATE” status by a triager by observing the differences between report submission times and the duplicate status assignment times. The same cannot be done for primaries because their time stamp represents the assignment to a developer in charge of the fix. If the report has not been assigned, it is left open. Furthermore, the assignment information for a primary report can only be obtained from auxiliary XML files that are not contained in the main problem report file. For this thesis, we chose a random sample of 1,000 primary reports to infer the time required to assign them to developers. In instances where a report is assigned and later reassigned, we only count the first time stamp. In cases where reports have not been assigned, we set the date equal to the date of the download.

Table 15 summarizes the distribution of time for manually triaging both primary and duplicate reports, with an approximate 95% confidence interval for the median of primaries for the entire data set. The lower and upper bounds of this interval are values from the ordered sample with ranks:

\[
\text{Lower Bound Ranked Value} : \frac{n}{2} - \frac{1.96\sqrt{n}}{2},
\]

\[
\text{Upper Bound Ranked Value} : 1 + \frac{n}{2} + \frac{1.96\sqrt{n}}{2},
\]

where \( n \) is the number of observations.

Minimum and maximum triage times are representative of a relatively small fraction of records where automation may not help much. For primaries, a time
close to the minimum might be achieved when a reporter self assigns the problem report resolution. Such cases are rare as most submitters do not have resolution assignment privileges. For duplicates, the minimum triage time may correspond to a report that directly references its primary. Times near the maximum might correspond to incomplete reports that have neither been assigned nor looked at by the triage team and likely will never be investigated by the developers. Consider the median triage time which is commonly used to represent typical values in such skewed-right distributions. If the triage team takes 18 days to assign a problem to a developer or 2 days to declare it a duplicate, a fast automated system could alleviate the backlog of reports.

Preprocessing of a newly submitted report to generate a predicted class takes approximately 2 minutes on a Dell Optiplex 960 with 2.8 GHz Intel i7 Quad Core processor, running with 8GB of RAM on Windows 7. Since the processing time for a single report is low, the time to build the model becomes more important. Building the largest retraining model took 25 minutes on the same Dell Optiplex 960. The retraining model only needs to be rebuilt every 2,500, 5,000, 10,000, 15,000 or 20,000 instances. These processing times are certainly feasible for any application in order to speed up triaging, reduce backlogs, and free up human resources for other project work.

4.12 Threats to Validity

The chosen subset of similarity measures can be considered a threat to internal validity. Our similarity measures represent two approaches: a word frequency based approach and a word sequence based approach. An approach that we did not try is topic modeling. However, topic modeling requires extensive dictionaries that may need to be updated frequently, as new reports and knowledge appear. This can be

<table>
<thead>
<tr>
<th></th>
<th>Primary</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>15 seconds</td>
<td>11 seconds</td>
</tr>
<tr>
<td>First Quartile</td>
<td>1 day</td>
<td>4 hours</td>
</tr>
<tr>
<td>Median</td>
<td>18 days</td>
<td>2 days</td>
</tr>
<tr>
<td>Third Quartile</td>
<td>229 days</td>
<td>45 days</td>
</tr>
<tr>
<td>Maximum</td>
<td>2330 days</td>
<td>3726 days</td>
</tr>
<tr>
<td>95% CI for Median</td>
<td>4.89 days - 47.7 days</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 15: Time to Manually Triage By Report Type
difficult when working with a data set of over 350,000 reports, which was the reason for exclusion. The chosen feature set representing each report can also be a threat to internal validity. The features were selected to describe both the document and the matches with other documents. Future work will explore larger feature sets as well as feature selection approaches that may improve classification importance.

The choice of the data set can be a threat to external validity. However, the three datasets chosen represent a wide range of products used by consumers. The Firefox dataset represents a popular web browser, the Open Office dataset represents a document management system, and finally the Eclipse dataset represents a software development environment. The Bugzilla bug tracking system is utilized by many companies. Analysis of other datasets is left as future work.

Bug tracking repositories are evolving ecosystems with bug reports changing status throughout time. As a result, it is likely that the problem reports from one snapshot of the repository may have different statuses when the same dataset is evaluated at a later date. Any research in bug tracking relies on the assumption that the ground truth is correct.

4.13 Conclusions

In this section we presented the framework for automatically labeling problem reports as primary or duplicate. We evaluated our framework on the Eclipse, Firefox and Open Office dataset. We demonstrated that a single similarity measure cannot allow us to effectively discern whether a report is primary or duplicate. Instead, we must utilize similarity measure, report and developer characteristics in conjunction with a classifier to automatically determine if a problem report is primary or duplicate.

Our results indicated that the training set size does not practically impact overall performance (despite being statistically significant). Thus, the framework can be deployed as early as needed. For the purpose of our work, we showed the system can be deployed after 2 months in Eclipse and 3 months in Firefox and Open Office. We showed that the quality of the reports significantly affects our performance when the dataset is itself problematic. Eclipse, with its well formed user base, consistently gave good results when using all reports and only relevant reports. In contrast, Firefox and Open Office performance improved dramatically when using the clean dataset.

Our work has been inspired by its real world relevance in assisting the triaging process. To the best of our knowledge, no automated triaging aide exists that can
discern between primary and duplicate reports in the context of a data set such as Eclipse. Researchers often point out that an automated tool is unnecessary as current search tools are sufficient for determining if a problem report is primary or duplicate. This makes the assumption that if a problem report exists outside the so called “top-20” suggested list, then it is a primary report. Such an assumption is incorrect, as the nature of the human language entails that matches can be found anywhere in the list of matches. Thus, the correct approach is to utilize the similarity scores in combination with the report and reporter characteristics to determine if a report is new or duplicate.

In the next set of sections, we will explore ways to determine the correct primary associated with each duplicate problem report.
5 Duplicate Classification - Group Centroid and Time Windows

Using the approach described in Section 4, we can determine whether an incoming problem report is primary or duplicate. Once a problem report has been classified as a duplicate, we must ascertain the correct primary report associated with it. Given the diversity of the human language, it is difficult to classify a duplicate report to its exact primary. Instead, we utilize a list of 20 most likely matches and allow the user to select the correct match. A success in this instance would be finding the correct primary within the top 20 list of matches. Given the sheer size of the datasets, for instance over 300,000 reports in Eclipse and close to 700,000 reports in Mozilla, attempting to match to all prior reports is not only time consuming, but also prone to generating false matches. The potential for false matches is exacerbated by the nature of the English language, wherein the frequency of common words can occlude the rarer words when using word frequency based methods. As a result we explore the feasibility of using a group centroid based word weighting scheme. Unlike the more traditional TF-IDF which weights each word based on the entire corpus of documents, the group centroid approach weights words in a group of similar documents by measuring their frequency of occurrence within the group. In this research we explore the characteristics of open source problem repositories to determine if the search space required can be reduced to improve similarity matches.

Our contributions to the domain include:

- Development of a novel time window based method for detecting the correct primary for a duplicate problem report
- Demonstrating the effectiveness of a group centroid based approach for word weighting over the traditional TF-IDF methodology

5.1 Firefox Dataset

Given the size of the datasets, we utilized a portion of the Firefox dataset to determine whether a group centroid based methodology was viable. All bug reports submitted to the Firefox repository are included from the inception of the project to June 2010. The total number of bugs in Firefox from its initial release till June 2010 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 were reported recently with a status "UNCONFIRMED". It appears this is a bug in Bugzilla itself that stores different bug reports with the same ID if their status is UNCONFIRMED. Removing these duplicate entries results in a total of 74,585 bugs. Table 16 provides the high level summary of the dataset.

<table>
<thead>
<tr>
<th></th>
<th>Total Number of Bugs</th>
<th>Number of Bugs (no Invalid)</th>
<th>Number of Duplicate Bugs</th>
<th>Total Number of Groups (more then one)</th>
<th>Number of stand-alone primaries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Number of Bugs</strong></td>
<td>85,665</td>
<td>74,585</td>
<td>25,888</td>
<td>6,408</td>
<td>48,697</td>
</tr>
<tr>
<td><strong>Number of Bugs (no Invalid)</strong></td>
<td></td>
<td>74,585</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number of Duplicate Bugs</strong></td>
<td></td>
<td>25,888</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total Number of Groups (more then one)</strong></td>
<td>6,408</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number of stand-alone primaries</strong></td>
<td>48,697</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 16: Bug Distribution in Firefox

5.1.1 Group Size

The primary bug report with its duplicates forms the group of bugs. Figure 24 shows the number of groups ordered by group size. 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 with duplicates in Firefox. These groups contain only the primary report and a duplicate. Table 17 shows the number of groups with sizes one to five and the corresponding percentage of duplicates. In Firefox, we found 60 different group sizes. The overwhelming majority of duplicates reside in small groups of 2 - 16 reports.

<table>
<thead>
<tr>
<th>Group Size</th>
<th>Nr of Groups</th>
<th>Nr of Reports</th>
<th>% of Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups of 1</td>
<td>48,697</td>
<td>48,697</td>
<td>65%</td>
</tr>
<tr>
<td>Groups of 2</td>
<td>3,464</td>
<td>6,928</td>
<td>9%</td>
</tr>
<tr>
<td>Groups of 3</td>
<td>1,129</td>
<td>3,387</td>
<td>4.5%</td>
</tr>
<tr>
<td>Groups of 4</td>
<td>554</td>
<td>2,216</td>
<td>3%</td>
</tr>
<tr>
<td>Groups of 5</td>
<td>327</td>
<td>1,635</td>
<td>2.2%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>62,845</strong></td>
<td></td>
<td><strong>84%</strong></td>
</tr>
</tbody>
</table>

Table 17: Group Size Distribution in Firefox

82
5.1.2 Time Interval between Consecutive Reports

The majority of duplicate reports are organized in small groups. In this section we analyze the time interval between the submission of consecutive duplicate reports within a group. Here we consider all 6,408 groups of duplicates that contain more than one report. Table 18 describes the accumulation of duplicate reports over a period of six days. The first column identifies 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, 16% of the groups (row 3, column 2) 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.

Figure 24: Group Size Distribution, y-axis presented in Log 2 scale
Table 18: Time Interval Between Consecutive Reports As a Cumulative Percentage of All Groups

The results from Table 18 can be viewed as the basis for probabilistic prediction when the next duplicate is likely to be filed. In simple terms, given a duplicate report we can say that there is a 16% chance in Firefox that a bug will receive the first duplicate report within the same day, or a 21% chance it will receive the first duplicate within 2 days.

Figure 25 further explains the time interval between the first and second report. This interval varies greatly. The majority of bugs receive the first duplicate during the first few days (or months) from the original report. However some receive the first duplicate surprisingly late. A single bug, shown at the far right end of the x-axis, was reported twice with the duplicate filled 3,635 days (10 years) after the original. In the early releases of Firefox, a user complained about search capability - Searching
results for the word “cat” would include words such as “catch”. A different user filed a duplicate report years later. The initial report was assigned Minor severity. This could be a case of developers ignoring minor bugs. One may, therefore, see that problem resolution dynamics impacts the number of duplicate reports.

5.2 Time Windows to Reduce Search Space

In general, the time interval between consecutive reports tends to be short. This finding can be used to limit the search for duplicates only to the most recently updated groups. In other words, we rank all the groups of duplicates based on the 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 [6] because for each prediction we first sort the groups chronologically and then consider only the groups inside the window. The window is of fixed size so that the number of reports considered for comparison remains nearly constant and so does the runtime.

Our analysis revealed that a list of 2,000 groups includes the group that will receive the next duplicate with an accuracy of 95%. This way we will restrict the search in the second phase of the experiment to an average of 8,000 reports. This is a significant reduction for the search space. We start the experiment with more than 30,000 reports in the repository and this number would increase as the experiment progresses. Having a limited search space will improve both the performance of IR algorithms and the runtime.

5.3 Experimental Framework

The goal of the experiment is to evaluate the potential of the proposed method for duplicate report reduction. In the context of Firefox, the data set contains 74,585 bug reports clustered in 6,408 groups with more than one report and 48,697 stand-alone primaries. We ordered bug reports chronologically, based on reporting date. For the purpose of this experiment, we consider the first 50% of bug reports to be historical and treat them as preexistent. We will evaluate our approach on the remaining 50%. The group that contains the most recently filed report (primary or a duplicate) is ranked on top of the initial list. Then we apply IR techniques to build the suggested list of potentially matching reports and present it to the reporter. 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., not reported before, we forward it into the database. This approach simulates the assumption that reporters or triagers never misjudge the new bug by reporting it as a variant of an existing one. We understand this is not a realistic assumption and discuss it later as one of the study’s validity threats.

Considering 50% of the reports as preexistent is not a limitation for our approach, rather it is a chosen experimental setup. It is possible to apply our approach as soon as we have more than 2,000 groups in the repository. As the number of the groups increases the benefits of our system become more pronounced.

We compare the following four approaches:

- TF/IDF only. Here we compare the vector representing a new report to every vector that is currently in the database. The 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. Here we applied the time window to limit the scope of groups under consideration for search. Only the reports within 2,000 groups are considered. These reports are weighted using TF/IDF. The scoring and building of the suggested list follows the same procedure as above.

- Time Window - Group Centroids. In the third approach we use the same time 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:
– 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 email, 0.33 function, 0.33 after, 0.33 enter, 0.33 recipient.

• TimeWindow - Group Centroids - TF/IDF. The fourth approach follows the centroid technique described above. However we weight each term in centroids using TF/IDF weighting scheme.

In the next section, we will compare how effective each of these four approaches are in terms of offering the reporter a correct problem report for matching.

5.4 Results
The results of performance evaluation experiments are presented in Figure 26. The third approach, using the time window and group centroid, yields the best recall rates reaching up to 53% with a list size of 20. Using the sliding window to limit the search has a clear impact on the classification performance. The first approach which searches all the bugs in the database not only has the lowest recall, but also the runtime increases with the number of reports in the database. On the contrary, the Time-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 approaches achieve similar performance. Using centroids weighted with TF/IDF has a slight advantage 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 approach uses as weights for each term the values calculated from the centroid of each group. On the other hand the fourth approach 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 [4] 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, 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.

5.5 Threats to Validity

Several assumptions underly the validity of our findings. We assume that users who take time to report problems in open source development environments are willing to assume a more proactive role in software quality improvement process. The recall rates reported above describe the presence of the matching problem report in the list of suggested reports. These recall rates offer the upper bound on the potential to reduce the number of duplicate reports following the described approach. We assume that the reporter will examine all the suggested reports and correctly detect the match. The total number of duplicates would be reduced by up to 53%, assuming the report list of size 20.

However, not all users may be willing to carefully examine the suggested-list. Impatience, the lack of time and a range of other reasons may cause a reporter to ignore the process and file a duplicate report. New and inexperienced users may not be able to correctly identify matching report in the suggested list [25]. On the other hand, some users may identify incorrect matches in the list, especially when the match does not exist. Unfortunately, our current experimental framework did not allow us to reliably estimate how many original bug reports would eventually end up incorrectly reported as comments in the existing but unrelated reports.

The time window selection was based on empirically evaluating the first 50% of the dataset. As the repository ages, a time window of 2,000 may not be adequate for generating correct matches. 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.

5.6 Conclusions

This chapter focused on improving the search features of bug tracking systems by helping reporters find similar, already reported bugs. The approach is intended to assist reporters, who face difficulties classifying duplicate reports. In this chapter, we analyzed the dynamics of duplicate report submission over the entire history of
Firefox project. We found empirical evidence that the time that elapses between the submission of related bug reports can be used to limit the search space for potential duplicate matches. As the duplicate search tool our technique achieves recall rates of up to 53%. This approach can improve search features of bug tracking systems and provides users with a short list of most “active” bugs to search through. User awareness of other reported bugs in the open source development increases through the examination of the suggested reports. At the same time, reporters are encouraged to add additional information into existing reports if they find the primary report in the list. However, it is possible that the suggested list does not contain the report the user is attempting to submit. Future work in this area would evaluate the likelihood of a user incorrectly selecting an existing report and its impact on duplicate propagation.

The group centroid weighting with cosine similarity approach assumes that the English, or any language, is a distribution of words. As humans we know that our language contains ordered structure. In the next section we will present a sequence matching technique called longest common subsequences that preserves word order. The approach has been utilized in gene sequencing and the Unix “diff” command, but has not seen use in the domain of document classification.
6 Duplicate Classification - Longest Common Subsequences

Much of the work in duplicate detection has leveraged word frequency based similarity measures. Such methods weight the occurrences of each individual word in the report and compare their frequency across all documents. These measures assign a lower weight to frequent words and reduce the possibility of false matches. Several classical techniques exist for weighting word frequencies, including TF/IDF and group centroids. Normalization of the language space is achieved through tokenization, stemming and stop word removal. The fundamental deficiency of word frequency based similarity measures is the lack of language context. A sentence in English, or any other language, is not a random clustering of words. Instead, it follows language structure rules with noun and verb phrases. While two documents may exhibit similar word frequencies, the issue described may not be similar and hence have a different context. As the size of the repository increases, so does the likelihood that two completely unrelated documents may be flagged as similar or duplicates.

The bioinformatics domain has been driving significant advances in string matching algorithms [38, 40], with successful applications of Rabin-Karp, Boyer-Moore, Needleman-Wunsch, Shift operations, and others. Ordered sequences play a critical role in bioinformatics, for example, the order of genes in sequences determines commonality [39].

Our major contributions to the domain of bug report duplicate detection is:

- Determining whether subsequence based methods can be used to measure similarity between problem reports.
- Applying a methodology that alleviate the loss of context/meaning issues faced by word frequency methods.
- Developing a novel weighting scheme that reduces the propensity for false matches due to differences in report size.

6.1 Motivation

One of the challenges with the word frequency based methods described earlier is that each approach works at a token level. The contextual information, that is the meaning, stored in each document is lost. Thus, due to the nature of the English
language false matches may be possible purely based on word frequency levels. To illustrate this we use the following two simple sentences as an example:

1. does report A describe a problem

2. report A does describe a problem

Report 1 is a question, Report 2 is a statement. As humans we are able to perceive this difference and would not classify them as similar. The question then arises, what would a machine perceive when using cosine similarity as its similarity measure. We describe the solution process in Table 19 below:

<table>
<thead>
<tr>
<th>Word</th>
<th>$freq(A)$</th>
<th>$freq(B)$</th>
<th>$A \cdot B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>does</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>report</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>describe</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>problem</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

$\|A\| = 2.45$  $\|B\| = 2.45$  $\cos(\theta) = 1$

Table 19: Worked Example of Cosine Similarity

A cosine similarity scheme would determine Report A and Report B to be identical to each each with a perfect score of 1. Cosine similarity is agnostic to word order, and the context or meaning of each document is lost. Thus, for large documents it is likely that false matches can be made due to the natural frequency of words in English, or any other language. As a more concrete example, let us consider two problem reports (175655 and 171777) from the Firefox repository. The text contained within each report, after performing tokenization, stemming, and stop word removal are as follows:

171777: bookmark separ misplac sidebar step reproduc 1 copi bookmarkshtml mozilla 12a applic data folder phoenix applic data folder 2 separ appear correctli pull-down menu manag bookmark window 3 view -> sidebar -> bookmark 4 separ ar all top bookmark pane between item separ still appear correctli pull-down menu manag bookmark window — adit test 5 open manag bookmark move separ 6 separ disappear bookmark pane correctli displai pull-down menu manag bookmark window 7 close bookmark view -> sidebar -> bookmark 8 separ appear top pane
Applying cosine similarity, the score is 0.68. In problem report systems, this score typically implies that these reports are each other’s duplicates. However, while they discuss issues pertinent to bookmarks, the actual problems are different. In the Firefox duplicate repository they are marked as distinct primary reports. The cosine similarity measure ignores one critical factor in documents written in natural language – the order of words. Therefore, a system that leverages not only the frequency of similar words but also the word ordering may create a more effective duplicate detection system.

Suppose, instead of taking the frequency of words we choose to find word positions in 175655 where it matches report 171777. We would obtain matches to report 171777 in the following word positions of 175655 (using a 0 starting number scheme): [3, 8, 23, 27, 29, 39, 42, 44, 45, 47, 53, 56, 58, 59, 62, 72, 79, 80, 109, 110, 116, 119, 124], or a chain of 23 matching words. Dividing this match score by the size of the matched document 171777 yields a similarity measure of 0.20. These two reports do not appear likely to be duplicates.

The shortcomings of cosine similarity are not isolated to the ordering of words. The technique cannot account for spelling errors, especially when they affect many of the words in the string. Typographical errors also negatively affect TF-IDF and Group Centroids, which assign a lower weight to common words and a higher weight to uncommon words. Words containing typos will be assigned higher weights as they are less frequent within the repository. Hence, documents containing common typos are more likely to match. The weighting schematic themselves raise challenges. The
centroid based approach adds new tokens as the diversity of language changes in groups of similar documents. Thus, in order for a new report to have an adequate match it must have a substantial portion of tokens and token distributions in common with the centroid of the group.

6.2 Firefox Dataset

The problem reporting data sets for our study come from Firefox and utilize the Bugzilla problem reporting system and have been often used in duplicate detection studies. The dataset consists of 25,045 duplicate problem reports.

In line with our prior work, we consider the first 50% of the dataset to be preexisting and empirically evaluate our approach on the latter 50%.

6.3 Longest Common Subsequences

The longest common subsequence (LCS) algorithm operates on a set of strings [46]. A subsequence is a sequence that can be derived by deleting some elements of the sequence without changing the order of the remaining elements. LCS has been applied to several domains. For example, in molecular biology DNA sequences can be represented as combinations of four basic blocks – A, C, G and T. As new gene sequences are found, biologists are interested in determining what subsequences are common with some known genes. While LCS has generally been applied at a character level, it can be easily modified to work at the word level.

The longest common subsequence between report 171777 and 175655 includes the following words: {bookmark, separ, folder, appear, separ, bookmark, correctli, manag, window, bookmark, dispai, menu, bookmark, bookmark, sidebar, separ, appear, separ, bottom, bookmark, dispai, bookmark} at a total length of 23 words. While the tokens in the LCS are important, we are more interested in finding the actual length of the LCS. A simple method for determining the length is described in Algorithm 1. In our experiments, we will use LCS function built in Perl within Algorithm::Diff library [47]. Our focus is not to improve the algorithm, but to leverage its characteristics to detect duplicate bug reports.
6.4 Duplicate Detection Methodology

Prior to the application of any duplicate classification method the text found in the title and summary fields of each problem report needs to be brought into a form that enhances the chance to correctly recognize duplicates.

6.4.1 Factor LCS Methodology

We consider Title, Summary, Classification, Component and Product fields of a Firefox bug report. While the Title has shown to be sufficient in word frequency based measures, it simply does not provide enough depth to be useful in generating common matches with LCS. Each report in the repository first goes through tokenization, stemming and stop word removal. Each report is compared to the ones submitted prior to it. LCS score is computed for each report pair.

Taking the highest LCS score as the indication of a match does not work well. Any short document is likely to have a higher LCS match scores with longer reports. Consequently, we weight the LCS score using Match Size Within Group Weight (MSWGW).

The MSWGW is defined as the sum of the ratios of the LCS value over the sizes of matching documents. The MSWGW is calculated as follows:

\[
MSWGW = \sum \left( \frac{C[m,n]^2}{m} \right) \times \sum \frac{C[m,n]}{N} \tag{1}
\]

where \(N\) is the number of reports within the group, \(C[m,n]\) is the number of words matched between report \(X\) of size \(m\) and report \(Y\) of size \(n\), and \(m\) is the size of the matching document \(Y\). For example, suppose Report A matches to reports B1, C1, D1 and E1 contained in duplicate groups X1, Y1, X1 and Z1, respectively, as shown in Table 20 below.

<table>
<thead>
<tr>
<th>Report</th>
<th>Grp</th>
<th>Matches</th>
<th>Size of Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>X1</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>C1</td>
<td>Y1</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>D1</td>
<td>X1</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>E1</td>
<td>Z1</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 20: Classifying Report A

We can compute the MSWGW values as follows:
\[ MSWGW(X_1) = \frac{(20^2/100 + 10^2/25) / 2 * (20 + 10)}{2} = 60 \]
\[ MSWGW(Y_1) = \frac{15^2/20 * 15/1}{1} = 168.75 \]
\[ MSWGW(Z_1) = \frac{10^2/20 * 10/1}{1} = 50 \]

Without MSWGW we may have considered report A to be a part of group X1. With MSWGW we classify it as a part of group Y1. MSWGW factor allows us to reduce false matches to large groups and large reports.

Additionally, we determined that the Classification (Class), Component (Comp) and Product (Prod) fields played a critical role in determining which primary report a duplicate was associated with. Reports describing similar problems had at least one of the three fields similar. Thus, the likelihood of a match is can be increased by boosting the MSWGW score when one or more of the fields match. We exemplify this phenomenon in Table 21 below.

<table>
<thead>
<tr>
<th>Rep</th>
<th>Grp</th>
<th>Comp (Rep)</th>
<th>Comp (Grp)</th>
<th>Class (Rep)</th>
<th>Class (Grp)</th>
<th>Prod (Rep)</th>
<th>Prod (Grp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X1</td>
<td>1</td>
<td>2</td>
<td>C1</td>
<td>C1</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>A</td>
<td>Y1</td>
<td>1</td>
<td>1</td>
<td>C1</td>
<td>C2</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>A</td>
<td>Z1</td>
<td>1</td>
<td>3</td>
<td>C1</td>
<td>C3</td>
<td>P1</td>
<td>P3</td>
</tr>
</tbody>
</table>

Table 21: Impact of Classification, Component and Product

From Table 21 we note that Group X1 has only one of the three fields (Comp, Class and Prod) similar to the test report. Group Y1 has two fields in common, while Group Z1 has only one field in common. Using the data in Table 21 we can boost the MSWGW scores when the Comp, Class and Prod values match by using the following logic. When the test report and the group have the same values we can multiply the MSWGW score by a specified factor value. If the two factor values do not match then we can leave the MSWGW score as is. Suppose we assign weights to W1, W2, and W3 of 2, 3 and 4 respectively.

The FinalMSWGW values would be as follows:
\[ FinalMSWGW(X_1) = 60 + 60 * 3 + 60 = 300 \]
\[ FinalMSWGW(Y_1) = 168.75 * 2 + 168.75 + 168.75 * 3 = 1181.25 \]
\[ FinalMSWGW(Z_1) = 50 + 50 + 50 = 150 \]

The LikelyGroup would be associated to group Y1 as it had the highest FinalMSWGW value. To generate the top 20 suggested list we simply take the highest 20 FinalMSWGW values and the groups they are associated with.
The W1, W2 and W3 weights associated with FinalMSWGW can be found using several techniques. We apply a pseudo Simulated Annealing (SA) method to optimize the weights [51]. The base case is set to the MSWGW results. Unlike a traditional SA where a model exists, we optimize until the best recall rate is achieved. Given the likelihood that an SA may not find an optimal solution, processing is terminated after 10,000 iterations. Testing with higher number of iterations did not yield significantly better results.

For the scope of this chapter we optimize to the first year of Firefox data. We obtain weight scores as follows (rounded to one decimal): W1=17.2, W2=20.0, and W3=17.8. The higher weight assigned to Classification is expected as it is a top level categorization within Bugzilla while Product and Component have nearly equal weights as they are second level categorizations. However, the weights associated are likely to be application and search space dependent.

Generating the MSWGW scores takes approximately 1.5 seconds on an AMD 3GHz machine with 1.75GB RAM. Computing the FinalMSWGW score, sorting and generating the LikelyGroup list takes approximately 0.5 seconds. Thus, a triager will be given a suggested top 20 list for each new incoming report in under 2 seconds. The size of the document also plays a critical role in the time taken to generate the final results. The median size of all reports found within the Firefox repository was 68 words, with 95% of all documents having less than 200 words.

6.5 Results

In line with previous work, FactorLCS returns a list of 20 reports most likely to be duplicates of the report the user is trying to submit.
Figure 27: Recall by Experiments (Firefox)

Figure 27 summarizes the results of the application of FactorLCS on the March 2012 merged version of Firefox repository. As mentioned before, we applied the algorithm to the repository version that includes only Firefox problem reports and the version with the duplicates associated with the core reports too. We also ran duplicate detection experiments with and without the time window (marked TW in Figure 27).
<table>
<thead>
<tr>
<th>Approach</th>
<th>Results</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Analysis [1]</td>
<td>Recall: ~50%</td>
<td>Firefox (&lt;06)</td>
</tr>
<tr>
<td>Bayesian learning, text categorization [14]</td>
<td>Predicted 30% of duplicates</td>
<td>Eclipse</td>
</tr>
<tr>
<td>Text similarity clustering [4]</td>
<td>Recall: 51% (top-20)</td>
<td>Firefox Feb 05 to Oct 05</td>
</tr>
<tr>
<td>NLP with execution information [3]</td>
<td>Recall: 67-93% (top-20)</td>
<td>Firefox Jan 04 to April 04</td>
</tr>
<tr>
<td>NLP with execution information [3]</td>
<td>Recall: 43-72% (top-20) w/ NLP</td>
<td>Firefox Jan 04 to April 04</td>
</tr>
<tr>
<td>NLP with execution information [5]</td>
<td>Recall: ~53% (top-20)</td>
<td>Firefox Apr 02 to Jul 07</td>
</tr>
<tr>
<td>Discriminative model trained via SVM [5]</td>
<td>Recall: ~70% (top-20)</td>
<td>Firefox Apr 02 to Jul 07</td>
</tr>
<tr>
<td>BM25F algorithm [9]</td>
<td>Recall: ~68% (top-20)</td>
<td>Mozilla Jan 10 to Dec 10</td>
</tr>
<tr>
<td>BM25F algorithm [9]</td>
<td>Recall: ~75% (top-20)</td>
<td>Eclipse Jan 08 to Dec 08</td>
</tr>
<tr>
<td>Time window with centroids [7]</td>
<td>Recall: ~53% (top-20)</td>
<td>All Firefox up to June 10</td>
</tr>
<tr>
<td>Topic Modeling [11]</td>
<td>Recall: ~84% (top-20)</td>
<td>Eclipse Jan 08 to Dec 08</td>
</tr>
<tr>
<td>Time window with FactorLCS [9]</td>
<td>Recall: ~68% (top-20)</td>
<td>All Firefox up to March 12</td>
</tr>
<tr>
<td>Time window with FactorLCS [9]</td>
<td>Recall: ~73% (top-20)</td>
<td>All Firefox up to March 12</td>
</tr>
</tbody>
</table>

Table 22: Comparison of Results

We offer Table 22 as a means to compare the performance of our approach to existing ones from the literature. From Table 22 we note that the recall rates obtained through FactorLCS match or outperform existing research results. Generally, FactorLCS with the time window (TW) typically offers a 5% performance improvement over related techniques. Performance increase is obtained in spite of the experimentation with the most recent version of an ever expanding Firefox problem report data set. The results of duplicate detection drop by 6% when applied to which includes primaries outside of Firefox, but related literature did not tackle such data set extension either. We speculate a margin of improvement between FactorLCS and competing algorithms would remain the same.
6.6 Threats to Validity

The representativeness of the datasets used in experiments is an external validity threat. We believe Firefox is large enough and complicated enough to affirm relevance of the study. Similar results are expected from other projects where duplicate reports are submitted in natural language. Nevertheless, open source projects and open problem reporting policies allow anyone, regardless of their experience and understanding, to submit a report. Such reporting, including the absence of technical information and in many cases very poor use of language, imposes a burden on maintenance personnel, compared to proprietary projects. From this perspective, external threat to validity is significant.

Within the context of Firefox repository, several duplicate groups had been merged, status of several reports has been changed over time. Some primary bug reports were found to be duplicates. Changing ground truth makes it difficult to perform comparative analysis of our results with existing research. However, evolution of software artifacts is a known dynamic. Freezing the snapshot of a repository just to be able to compare performance of research algorithms does not seem to be the desirable approach either, except in cases where research community may organize benchmarks.

The next validity threat is related to the adequacy of the proposed technical approach. The subsequence based matching approach (LCS) relies on the presence of large bodies of text. When we used only the Title of reports to measure similarity, we failed to yield measurably improvements over existing methods. Titles are simply too short. Word frequency approaches are typically limited to the analysis and comparison of problem report titles only. We did not pursue detailed measurements of processing time, but having implemented both types of algorithms, it does not appear that LCS approach is prohibitively slower.

Lastly, the weights associated with LikelyGroup are likely to be application and search space dependent. For the scope of this chapter the weights are generalized across both applications using only the first year of Firefox data for optimization purposes. We are currently investigating the impact of the application as well as the search space required to determine the weights.
6.7 Conclusions

Duplicate detection in bug tracking systems is not a new problem. Bugzilla itself employs Boolean full text search engine. Bugzilla and many prior research approaches are based on word frequency counts, which fail to account for the context of the report.

This chapter represents the first effort at applying a string matching approach to duplicate bug report detection. Our results exceed the recall rates of prior reported research approaches by at least 5% while using more recent release of Firefox bug report repository. We show that longest common subsequences, and possibly other bioinformatics algorithms can be used to effectively detect duplicate reports. We also demonstrate the significance of proper primary bug report identification by demonstrating the long term impact of merging report branches, once they are recognized to represent the same underlying problem.

Natural language plays a critical role in the ability for systems to detect duplicates. Duplicate reports are submitted by users writing in British and American English. Typographical and alternate spellings are prevalent in duplicate reports. This raises an interesting area of future work. The breadth of the language in software problem reporting is a small subset of English. The prevalent terms in bug reports are very specific to the application under study. We believe that the development of application specific dictionaries may offer the new types of analysis that may improve report matching process. Such an approach would further normalize individual reports and allow increased effectiveness of duplicate report detection and classification algorithms.

While longest common subsequences solved the word frequency problem, it created a new challenge. While we are able to retain document structure, we now require distinct humans to share similar writing styles. As humans, we are well aware of the diversity of our language. In the next chapter we will explore a methodology that can combine a set of document similarity measures. Thus, we no longer rely on a single approach, but instead use a consensus based approach that relies on the viewpoint of multiple similarity measures.
7 Duplicate Classification - Multi-Label Classification

Traditional document classification techniques attempt to classify new documents to existing classes. But research indicates that real world documents seldom belong to a singular class. In [52] the authors applied a KNN classifier and LSI (Latent Semantic Indexing) on the Reuters news report data set. 9,603 news stories were categorized into one or more of 118 topics. In software engineering, similar multilabel classification techniques can be used to assign problem reports to one of many project areas, e.g. GUI, OS, Security, etc [53].

We propose a fusion framework that utilizes multi-label classification to address the duplicate classification problem. Our observations on existing document similarity measures indicate that no single approach exists that can perfectly classify problem reports. Moreover, our investigation of similarity measures demonstrates that some measures are better at classifying documents in certain portions of the dataset. Our approach leverages multi-label classification, and allows us to assign one or more similarity measures to each problem report. In other words, each problem report similarity measure (group centroid with cosine similarity, longest common subsequence, etc.) may assign the new report to a different group of previously reported problems.

Our contributions to the domain of duplicate problem report classification are as follows:

- Demonstrating how multi-label classification can improve weak problem report classification techniques.
- Demonstrating the effect of problem report categorization on overall recall rates.
- Demonstrating the effect of data set size on overall recall rates.
- Demonstrating the effect of duplicate group size on problem report similarity measures.
- Demonstrating how the lack of consensus (measured as the number of measures indicating a report was matched) affects the performance of the multilabel classifier.
- Demonstrating the effect of list size on overall recall rates.
7.1 Challenges in Existing Document Similarity Measures

The three most commonly used methodologies for measuring document similarity are: frequency, sequence and topic modeling based approaches. Frequency based approaches infer similarity by measuring how well the closeness of the frequency distribution of words across two documents. One of the most commonly used approaches, cosine similarity, is a vector space model that measures the cosine angle between the two chosen documents. Two documents with the same set of words appearing at the same frequency would have a cosine angle of 0, or a similarity score of 1. Two documents with no words in common would have a cosine angle of 90, or a similarity score of 0. The nature of the English language, or any language, entails that some words occur more frequently than others, for instance “the”, “be”, “to”, “of” and “and” are considered the 5 most frequent words in English. As a result, in very large documents, it is likely that measuring the frequency of words alone will cause false high matches. One commonly used approach is to exclude meaningless words that are common across all reports by utilizing stop word lists or word weighting schemes. However, the underlying principle of word frequency based methods considers similarity based on the frequency of words as opposed to the structure of sentences. Thus, while new techniques such as BM-25F have shown promise, they do not adequately represent how humans perceive language.

In order to retain document structure, one can apply sequence based methodologies that measure similarity by finding the largest ordered subset of words (or characters) in common between two documents. Two of the most commonly used methodologies are longest common subsequence and longest common substring. The former allows for words to be skipped, while the latter is more restrictive and require the same subset of unskipped ordered words in both documents. Subsequence and substring matching raises its own challenge. While it adequately addresses the order of words problem, it requires two distinct individuals to have the same writing style and use the same set of words to describe their thought process. Given the diversity within the human language, it is likely that individuals will have very different writing styles.

Topic modeling has emerged as a viable solution to address the frequency and sequence based approach limitations. Topic modeling assumes that a corpus of documents consists of a set of words, commonly called the vocabulary. Each document in turn consists of a set of topics, and each topic itself contains a set of words. Thus,
it is assumed that each distinct topic will have a particular set of words that occur
more frequently within it than in other topics. Topic modeling alleviates the issues
of frequency and sequence based methods by allowing a set of related words to form
a topic, as opposed to using frequency or sequence to measure similarity. However,
topic modeling itself presents several challenges. The vocabulary cannot be defined
at the start of a project, and must be updated continually as new knowledge enters
the system. Next, the number of topics and number of words per topic must also
be defined. Selecting too few topics will result in new reports being misclassified,
selecting too many topics will slow the classification procedure and cause duplicates
to be misclassified as new.

7.2 Motivation

We first demonstrate how one similarity measure can outperform another in different
problem reports. We intentionally choose the Eclipse dataset as an example to illus-
trate the fact that this phenomenon is not isolated to Firefox and exists across many
repositories. Our approach originated from the observation that certain similarity
measures outperform each other in different instances of problem report comparison.
The following example illustrates this phenomenon. The following problem report
#216547 was submitted to the Eclipse problem repository and was categorized as
a duplicate by the triager. This report and its associated primary report #216230
contain the following textual information:

<table>
<thead>
<tr>
<th>Problem report ID: 216547</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title: [Regression] One of the chart is missing when preview as POSTSCRIPT [1201]</td>
</tr>
<tr>
<td>Bug Status: Duplicate of bug 216230</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem report ID: 216230</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title: [Automation][Regression]Images from the resource folder disappears when view report as POSTSCRIPT</td>
</tr>
<tr>
<td>Bug Status: Primary</td>
</tr>
</tbody>
</table>

When using the Centroid weighting with Cosine Similarity measure, the duplicate
is matched to its primary with a score of 0.272. The match appears at position
34 and falls outside the top-20 list of potential primary reports suggested by the
measure. The top-20 suggested list is a commonly used method for providing a
list of 20 most likely matches to the triager. Therefore, the triager would likely assume that the report did not match with any existing report and categorize it as a primary. However, when using Longest Common Subsequence it is matched to its correct primary with a score of 0.33 at position 8. Thus, in this example Longest Common Subsequence would be the preferred measure for classifying reports similar to #216547.

To better understand this phenomenon we analyze the Eclipse 2008 dataset. The Eclipse 2008 dataset consists of all problem reports submitted in the year 2008. In Table 23, we highlight some measures and their recall rates when they are used in isolation on the Eclipse 2008 dataset. The recall rate is defined as the ratio of the correctly classified duplicate reports over the total number of duplicate reports. The Recall Rate is presented as a top-20 list, that is the suggested correct match lies within the top-20 possible matches.) Next, we assume that a perfect oracle exists that can determine which problem report classification measure works best on each report and uses that measure for that report. These ideal results obtained by combining two measures are shown in the Combined Recall column.

<table>
<thead>
<tr>
<th>Similarity Measure</th>
<th>Recall</th>
<th>Combined Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid</td>
<td>63%</td>
<td>69%</td>
</tr>
<tr>
<td>LCS</td>
<td>56%</td>
<td></td>
</tr>
<tr>
<td>Centroid w/ Factor</td>
<td>74%</td>
<td>80%</td>
</tr>
<tr>
<td>LCS w/ Factor</td>
<td>69%</td>
<td></td>
</tr>
<tr>
<td>All Four Measures Combined</td>
<td>82%</td>
<td></td>
</tr>
</tbody>
</table>

Table 23: Multi-Label Classification (Eclipse 2008)

Using this method and merging LCS and Centroid measures, we can improve our overall recall rate to a theoretical maximum of 69%, an improvement of 6-13% over the base measures. Similarly, being able to select between LCS with Factor and Centroid with Factor we can achieve a theoretical maximum of 80%, an improvement of 6-11% over the base measures. If a perfect oracle existed that could select the best of the four measures for each problem report, we could achieve a theoretical best recall rate of 82%.

The effectiveness of similarity measures also depends on the duplicate group size. To investigate this issue in more detail, we compared the performance of the Group Centroid with Cosine similarity measure to the Longest Common Subsequence measure as the number of reports in duplicate groups increases in the Firefox dataset.
Figure 28: Recall Rates for Different Similarity Measures as Repository Size Grows Using Firefox

Figure 28 illustrates the impact of the group size increases on recall rate. The group size is the number of reports associated with the current duplicate problem report. It is based on a snapshot of the current state of the repository. We found that the centroid measure that utilizes only the report’s title drops in performance as the duplicate group size increases. This behavior is expected, as the title contains a limited number of words and distinct users may use alternate words to describe the same issue. As the group size increases, each incoming report must have the same distribution of words as the duplicate group in order to yield a proper match.

In contrast, the Longest Common Subsequence (LCS) approach improves as the group size increased. This is once more due to the nature of this similarity measure. Unlike the Centroid measure in which each incoming report is matched to a group of reports, the LCS approach matches each incoming report to prior reports (primary or duplicate). As a result, an incoming report can match to any of the reports in a group.

Thus, one can appreciate the need for an approach that does not deteriorate in performance as the dynamics of the dataset changes.
7.3 Dataset Description

We utilize 4 different repositories to determine the value of multi-label classification. The datasets are defined as follows:

- Eclipse w/ Group Centroid: we utilize the complete Eclipse repository and perform similarity matches on the nearest 70,000 groups of reports.

- Firefox w/ Group Centroid: we utilize the complete Firefox repository and perform similarity matches on the nearest 20,000 groups of reports. Primary reports from the Core development chain are included to demonstrate the reduction in performance when using factors.

- Firefox: we utilize the complete Firefox repository and perform similarity matches to all reports submitted prior to the test report. This creates a framework that is more realistic, as the group information cannot be assumed without having a triager go through and assess each problem report.

- Open Office: we utilize the complete Open Office repository and perform similarity matches to all reports submitted prior to the test report. This creates a framework that is more realistic, as the group information cannot be assumed without having a triager go through and assess each problem report.

<table>
<thead>
<tr>
<th></th>
<th>Eclipse w/ Group Centroid</th>
<th>Firefox w/ Group Centroid</th>
<th>Firefox</th>
<th>OpenOffice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number of Duplicates</td>
<td>41,748</td>
<td>31,034</td>
<td>31,034</td>
<td>19,842</td>
</tr>
<tr>
<td>Total Number of Detectable Duplicates</td>
<td>41,748</td>
<td>25,085</td>
<td>25,085</td>
<td>19,842</td>
</tr>
<tr>
<td>Duplicates Within Nearest N Groups</td>
<td>39,440 (70,000)</td>
<td>24,255 (20,000)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 24: Characteristics of the Eclipse, Firefox and Open Office Repositories

Table 24 summarizes the characteristics of each repository. The discrepancy between the “Total Number of Duplicates” and “Total Number of Detectable Duplicates” in Firefox is due to Mozilla’s software development ideology. Mozilla’s main code development (called Core) spawns various products such as Firefox, Bugzilla, etc. Due to code-sharing between Core and Firefox, some primaries for Firefox may
exist in the Core problem report repository. A user may report a Firefox problem that is actually caused by an issue from the Core system, i.e., outside the Firefox repository. Some Firefox bug reports are tagged as duplicates of a primary report in the Core. If the duplicate report is the first in a series of duplicates associated with the Core problem, an automated method cannot find the matching primary because it is in different repository.

The "Duplicates Within Nearest N Groups" indicates the number of duplicate reports (out of the detectable reports) whose closest match was within the 20,000 (Firefox) and 70,000 (Eclipse) most recently updated groups.

7.4 Baseline Results

In Table 25, 26, 27 and 28 we present the baseline duplicate classification results for each dataset using a variety of similarity measures. These results offered us an understanding of the potential of our approach. The notation in the Method column of Table 25, 26, 27 and 28 is explained below.

- CS and LCS – refer to Cosine Similarity and Longest Common Subsequence similarity measures.
- Suffix “win X” refers to the window size being changed to X.
- Suffix “w/factor” means that the similarity scores were weighted based on factor similarity between the matched report and the test report, as described earlier.
<table>
<thead>
<tr>
<th>Method</th>
<th>Eclipse</th>
<th>MULAN</th>
<th>Method</th>
<th>Eclipse</th>
<th>MULAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid</td>
<td>38%</td>
<td>+26%</td>
<td>Centroid</td>
<td>50%</td>
<td>+14%</td>
</tr>
<tr>
<td>Centroid w/factor</td>
<td>47%</td>
<td>+22%</td>
<td>Centroid w/factor</td>
<td>57%</td>
<td>+7%</td>
</tr>
<tr>
<td>LCS</td>
<td>42%</td>
<td>+17%</td>
<td>LCS</td>
<td>42%</td>
<td>+22%</td>
</tr>
<tr>
<td>LCS w/factor</td>
<td>51%</td>
<td>+13%</td>
<td>LCS w/factor</td>
<td>33%</td>
<td>+11%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>23%</td>
<td>+41%</td>
<td>Centroid (win 1000)</td>
<td>26%</td>
<td>+38%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>25%</td>
<td>+30%</td>
<td>Centroid (win 1000)</td>
<td>28%</td>
<td>+36%</td>
</tr>
<tr>
<td>LCS (win 1000)</td>
<td>23%</td>
<td>+41%</td>
<td>LCS (win 1000)</td>
<td>25%</td>
<td>+39%</td>
</tr>
<tr>
<td>LCS (win 1000) w/factor</td>
<td>26%</td>
<td>+38%</td>
<td>LCS (win 1000) w/factor</td>
<td>28%</td>
<td>+36%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>27%</td>
<td>+37%</td>
<td>Centroid (win 2000)</td>
<td>31%</td>
<td>+33%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>30%</td>
<td>+34%</td>
<td>Centroid (win 2000)</td>
<td>34%</td>
<td>+30%</td>
</tr>
<tr>
<td>LCS (win 2000)</td>
<td>27%</td>
<td>+37%</td>
<td>LCS (win 2000)</td>
<td>29%</td>
<td>+35%</td>
</tr>
<tr>
<td>LCS (win 2000) w/factor</td>
<td>31%</td>
<td>+33%</td>
<td>LCS (win 2000) w/factor</td>
<td>33%</td>
<td>+31%</td>
</tr>
</tbody>
</table>

Table 25: Baseline Duplicate Report Classification Performance on Eclipse

<table>
<thead>
<tr>
<th>Method</th>
<th>Firefox</th>
<th>MULAN</th>
<th>Method</th>
<th>Firefox</th>
<th>MULAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid</td>
<td>50%</td>
<td>+22%</td>
<td>Centroid</td>
<td>64%</td>
<td>+8%</td>
</tr>
<tr>
<td>Centroid w/factor</td>
<td>34%</td>
<td>+38%</td>
<td>Centroid w/factor</td>
<td>52%</td>
<td>+20%</td>
</tr>
<tr>
<td>LCS</td>
<td>56%</td>
<td>+16%</td>
<td>LCS</td>
<td>52%</td>
<td>+20%</td>
</tr>
<tr>
<td>LCS w/factor</td>
<td>56%</td>
<td>+16%</td>
<td>LCS w/factor</td>
<td>38%</td>
<td>+34%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>43%</td>
<td>+20%</td>
<td>Centroid (win 1000)</td>
<td>48%</td>
<td>+24%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>33%</td>
<td>+39%</td>
<td>Centroid (win 1000)</td>
<td>34%</td>
<td>+38%</td>
</tr>
<tr>
<td>LCS (win 1000)</td>
<td>43%</td>
<td>+29%</td>
<td>LCS (win 1000)</td>
<td>42%</td>
<td>+30%</td>
</tr>
<tr>
<td>LCS (win 1000) w/factor</td>
<td>44%</td>
<td>+28%</td>
<td>LCS (win 1000) w/factor</td>
<td>43%</td>
<td>+29%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>49%</td>
<td>+23%</td>
<td>Centroid (win 2000)</td>
<td>56%</td>
<td>+16%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>35%</td>
<td>+37%</td>
<td>Centroid (win 2000)</td>
<td>38%</td>
<td>+34%</td>
</tr>
<tr>
<td>LCS (win 2000)</td>
<td>49%</td>
<td>+23%</td>
<td>LCS (win 2000)</td>
<td>48%</td>
<td>+24%</td>
</tr>
<tr>
<td>LCS (win 2000) w/factor</td>
<td>50%</td>
<td>+22%</td>
<td>LCS (win 2000) w/factor</td>
<td>48%</td>
<td>+24%</td>
</tr>
</tbody>
</table>

Table 26: Baseline Duplicate Report Classification Performance on Firefox w/ Group Centroids
### Table 27: Baseline Duplicate Report Classification Performance on Firefox

<table>
<thead>
<tr>
<th>Method</th>
<th>Firefox</th>
<th>MULAN</th>
<th>Method</th>
<th>Firefox</th>
<th>MULAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid</td>
<td>48%</td>
<td>+20%</td>
<td>Centroid</td>
<td>58%</td>
<td>+10%</td>
</tr>
<tr>
<td>Centroid w/factor</td>
<td>49%</td>
<td>+19%</td>
<td>LCS</td>
<td>57%</td>
<td>+11%</td>
</tr>
<tr>
<td>LCS</td>
<td>52%</td>
<td>+16%</td>
<td>LCS</td>
<td>49%</td>
<td>+19%</td>
</tr>
<tr>
<td>LCS w/factor</td>
<td>53%</td>
<td>+15%</td>
<td>LCS w/factor</td>
<td>49%</td>
<td>+19%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>25%</td>
<td>+43%</td>
<td>Centroid (win 1000)</td>
<td>27%</td>
<td>+41%</td>
</tr>
<tr>
<td>Centroid (win 1000) w/factor</td>
<td>25%</td>
<td>+43%</td>
<td>Centroid (win 1000) w/factor</td>
<td>28%</td>
<td>+40%</td>
</tr>
<tr>
<td>LCS (win 1000)</td>
<td>24%</td>
<td>+44%</td>
<td>LCS (win 1000)</td>
<td>24%</td>
<td>+44%</td>
</tr>
<tr>
<td>LCS (win 1000) w/factor</td>
<td>25%</td>
<td>+43%</td>
<td>LCS (win 1000) w/factor</td>
<td>25%</td>
<td>+43%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>30%</td>
<td>+38%</td>
<td>Centroid (win 2000)</td>
<td>34%</td>
<td>+34%</td>
</tr>
<tr>
<td>Centroid (win 2000) w/factor</td>
<td>31%</td>
<td>+37%</td>
<td>Centroid (win 2000) w/factor</td>
<td>34%</td>
<td>+34%</td>
</tr>
<tr>
<td>LCS (win 2000)</td>
<td>29%</td>
<td>+39%</td>
<td>LCS (win 2000)</td>
<td>30%</td>
<td>+38%</td>
</tr>
<tr>
<td>LCS (win 2000) w/factor</td>
<td>30%</td>
<td>+38%</td>
<td>LCS (win 2000) w/factor</td>
<td>30%</td>
<td>+38%</td>
</tr>
</tbody>
</table>

### Table 28: Baseline Duplicate Report Classification Performance on Open Office

<table>
<thead>
<tr>
<th>Method</th>
<th>Open-Office</th>
<th>MULAN</th>
<th>Method</th>
<th>Open-Office</th>
<th>MULAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centroid</td>
<td>43%</td>
<td>+18%</td>
<td>Centroid</td>
<td>52%</td>
<td>+9%</td>
</tr>
<tr>
<td>Centroid w/factor</td>
<td>46%</td>
<td>+15%</td>
<td>Centroid w/factor</td>
<td>31%</td>
<td>+30%</td>
</tr>
<tr>
<td>LCS</td>
<td>46%</td>
<td>+15%</td>
<td>LCS</td>
<td>42%</td>
<td>+19%</td>
</tr>
<tr>
<td>LCS w/factor</td>
<td>47%</td>
<td>+14%</td>
<td>LCS w/factor</td>
<td>46%</td>
<td>+15%</td>
</tr>
<tr>
<td>Centroid (win 1000)</td>
<td>31%</td>
<td>+31%</td>
<td>Centroid (win 1000)</td>
<td>36%</td>
<td>+25%</td>
</tr>
<tr>
<td>Centroid (win 1000) w/factor</td>
<td>33%</td>
<td>+28%</td>
<td>Centroid (win 1000) w/factor</td>
<td>31%</td>
<td>+30%</td>
</tr>
<tr>
<td>LCS (win 1000)</td>
<td>30%</td>
<td>+31%</td>
<td>LCS (win 1000)</td>
<td>33%</td>
<td>+28%</td>
</tr>
<tr>
<td>LCS (win 1000) w/factor</td>
<td>33%</td>
<td>+28%</td>
<td>LCS (win 1000) w/factor</td>
<td>34%</td>
<td>+27%</td>
</tr>
<tr>
<td>Centroid (win 2000)</td>
<td>36%</td>
<td>+25%</td>
<td>Centroid (win 2000)</td>
<td>42%</td>
<td>+19%</td>
</tr>
<tr>
<td>Centroid (win 2000) w/factor</td>
<td>38%</td>
<td>+23%</td>
<td>Centroid (win 2000) w/factor</td>
<td>33%</td>
<td>+26%</td>
</tr>
<tr>
<td>LCS (win 2000)</td>
<td>35%</td>
<td>+26%</td>
<td>LCS (win 2000)</td>
<td>38%</td>
<td>+23%</td>
</tr>
<tr>
<td>LCS (win 2000) w/factor</td>
<td>37%</td>
<td>+24%</td>
<td>LCS (win 2000) w/factor</td>
<td>39%</td>
<td>+22%</td>
</tr>
</tbody>
</table>
7.5 Best Achievable Results

The previous section provided the baseline results when similarity measures are used one at the time. We next determine the best results that we can hope to achieve using a “perfect” combination classifier. For each problem report, this perfect classifier would select only those measures that correctly identify the primary report in their top-20 list. If such an oracle existed, it would provide the best performance we can possibly achieve using this approach. Such a measure also provides a benchmark performance that we can aim to achieve when building combination classifiers automatically.

<table>
<thead>
<tr>
<th>Can Be Classified By</th>
<th>Eclipse</th>
<th>Firefox w/ Group Centroids</th>
<th>Firefox</th>
<th>OpenOffice</th>
</tr>
</thead>
<tbody>
<tr>
<td>No measure</td>
<td>22.31%</td>
<td>11.14%</td>
<td>16.31%</td>
<td>18.23%</td>
</tr>
<tr>
<td>Only 1</td>
<td>4.76%</td>
<td>3.39%</td>
<td>3.60%</td>
<td>5.74%</td>
</tr>
<tr>
<td>2-10</td>
<td>45.17%</td>
<td>49.51%</td>
<td>49.56%</td>
<td>39.52%</td>
</tr>
<tr>
<td>11-23</td>
<td>18.36%</td>
<td>25.00%</td>
<td>21.14%</td>
<td>26.67%</td>
</tr>
<tr>
<td>All Measures</td>
<td>9.39%</td>
<td>10.97%</td>
<td>9.39%</td>
<td>9.84%</td>
</tr>
<tr>
<td>Total</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 29: Proportion of duplicates correctly matched by similarity measures

To determine the best achievable duplicate classification results we analyze which similarity measures, we have 24 total in Table 26, can correctly match each duplicate problem report to its primary. Table 29 shows the number of measures that correctly classify all problem reports. The row “No measure” indicates that none of the 24 measures was able to match these problem report to their primaries.

7.6 Naive Fusion Method

Fusion of match scores is often utilized in biometrics [58] to combine scores from various modalities. We apply the SUM and MAX rule to determine if a naive fusion scheme can outperform the single similarity metric approach. The three rules are defined as follows:

- SUM rule - when two or more measures suggest the same report we add their scores;
- MAX rule - when two or more measures suggest the same report we take the maximum score.
Given the different ranges in the similarity scores, [0-1] for methods without factors and [0-4] for methods with factors, we apply a normalization scheme as follows:

- **Raw Score** - all similarity scores are left as-is
- **Max Adjusted** - all similarity scores in a method are divided by the highest score for the method
- **01Normalized** - all factor based scores are divided by 4 to create values in the [0-1] range for both factor and non factor based methods

The results of the naive fusion scheme are shown in Table 30. Results for Eclipse, Firefox and Open Office are shown. Due to data loss, the naive scheme could not be applied to the Firefox with Group Centroids dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Fusion Type</th>
<th>Raw Score</th>
<th>Max Adjusted</th>
<th>01Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>SUM</td>
<td>65.14%</td>
<td>60.40%</td>
<td>62.86%</td>
</tr>
<tr>
<td></td>
<td>MAX</td>
<td>59.08%</td>
<td>46.35%</td>
<td>50.14%</td>
</tr>
<tr>
<td>Firefox</td>
<td>SUM</td>
<td>65.96%</td>
<td>61.04%</td>
<td>65.88%</td>
</tr>
<tr>
<td></td>
<td>MAX</td>
<td>59.96%</td>
<td>46.44%</td>
<td>59.62%</td>
</tr>
<tr>
<td>Open Office</td>
<td>SUM</td>
<td>61.96%</td>
<td>60.84%</td>
<td>62.70%</td>
</tr>
<tr>
<td></td>
<td>MAX</td>
<td>48.88%</td>
<td>41.27%</td>
<td>52.66%</td>
</tr>
</tbody>
</table>

Table 30: Naive Fusion Scheme Results

The naive fusion scheme illustrates the benefits of using more than one document similarity measure. The net improvement over a single method are 8%, 8% and 11% for Eclipse, Firefox and Open Office respectively. The MAX rule in all instances gave a worse result than the SUM rule. The MAX rule is prone to being confounded by outliers, while the SUM rule seeks out repeated matches across many similarity measures.

### 7.7 Multi-Label Classification

Multi-label classification analyzes each problem report and predicts which similarity measures (out of the pool of 24) assign it to the correct primary. To achieve this goal we used MULAN, a multi-label classification scheme developed on top of the Weka machine learning software package[54].
7.7.1 Model Training in MULAN

As with any other supervised learner, MULAN requires training data to create the model. The training data consists of a set of attribute values for every problem report used for training. The attributes used to characterize each problem report are described in Tables 9, 10 and 11. Additionally, unlike a traditional classifier, the MULAN label format is a vector of Boolean values. Each value indicates whether the specific measure correctly detected matching primary (1/0). A value of 1 indicates that a specific method is able to detect the duplicate in the top 20 list, a value of 0 indicates it is unable to do so. A set of 24 Boolean values represents the 24 methods.

7.7.2 Classification

Once the model is trained, the same format for feature vector is used in classification, except that “Label” attribute becomes the dependent (predicted) vector. When selecting the similarity measures for each problem report the output vector contains three values for each measure:

- Bipartition – is either TRUE or FALSE and it indicates whether the measure is predicted to correctly classify the problem report.

- Confidence – is a score between 0 and 1. It indicates the confidence level of MULAN in assigning the “Bipartition” value.

- Ranking – displays the rank of the “Confidence” value for the corresponding measure. It ranges from 1 (the highest confidence amongst all measures) to N (the lowest).

7.7.3 Generating the Unified Top-20 List

The final step in our multi-label classification approach is to generate the list of 20 reports as possible matches for a given problem report. Each similarity measure selects 20 most likely matches for the new report. By selecting more than one similarity measure, combining their top 20 match picks would create a list too long for the triager to check. Therefore, an automated procedure is needed to “merge” these lists.

To generate the top-20 list for every report, we first select similarity measures that MULAN assigns a “Bipartition” value of TRUE. Some of their lists may suggest
the same reports, but many are likely to be different. We use a fusion rule to generate
the final score. The two fusion rules we applied are the MAX rule and the SUM rule.
As the names indicate, the MAX rule takes the maximum match score when two or
more measures suggest the same report. The SUM rule adds the match scores when
two or more measures suggest the same report. In case only one similarity measure
selected a specific matching report, its match score is passed unchanged. The final
scores are then sorted in descending order and the 20 reports with the highest score
are presented to the triager.

7.8 Experimental Setup

We performed a series of experiments to test the performance of the described multi-
label classifier. The following three parameters were varied to create different ex-
perimental setups: the training procedure, the fusion process, and the confidence
measure weights. The entire experimental procedure is summarized in Figure 29.

7.8.1 Training Process

Two training processes were used, single instance training and retraining:

- Single Instance Training: the classifier was trained once using 200 (Firefox w/
  Group Centroids only), 250, 500, 1000, and 2500 reports from the start of the
dataset

- Retraining: the classifier was trained using the first 100 (Firefox w/ Group
  Centroids only), 250, 500, 1000 and 2500 reports and then tested on the next
batch of 100 (Firefox w/ Group Centroids only), 250, 500, 1000 and 2500
reports. The ground truth values are used to correct the classification errors
and the classifier is retrained for the next round.

7.8.2 Fusion Process

As mentioned earlier, we use the following fusion methods in the creation of the
top-20 match suggestions list:

- SUM rule - when two or more measures suggest the same report we add their
  scores;

- MAX rule - when two or more measures suggest the same report we take the
  maximum score.
7.8.3 Confidence Measure Weight

For every similarity measure that MULAN deems correct, it offers a Confidence score to indicate its certainty in selecting that measure. Each of these measures offers a similarity score for each of the twenty problem reports that they suggest as possible duplicate matches. These 20 scores can be normalized in five different ways. The normalization process is used to determine not only whether the MULAN confidence score has an impact, but also ensure that a single method does not overshadow others with a high similarity score.

- M1: Normalize the scores by dividing by the highest score for that particular measure. For example, suppose the Centroid measure yielded a match of 0.5. If this score is the highest match amongst all 20 reports suggest by Centroid, then we can divide by 0.5 to normalize the score to 1.0.
Figure 30: MULAN Results with Eclipse and No Retraining

- **M2**: Normalize the scores by dividing by the highest score for the particular measure and multiplying by the MULAN confidence score.

- **M3**: Normalize scores to a value between 0 and 1 by dividing the factor scores by 4. The factor scores are obtained by using the Factor weighting scheme described earlier. The highest possible score for a factor based score is 4 (when Product, Component and Classification match), as the highest base similarity score can be 1 (i.e. a 100% match).

- **M4**: Keep all match scores with their original values (that is between 0 and 1 for base measures and between 0 and 4 for factor measures).

- **M5**: Keep all match scores with their original values as in M4 and multiply by MULAN’s confidence score.
7.9 Results

The results of the multilabel classification process are provided in Figures 30, 31, 32, 33, 34, 35, and 36 for Eclipse No Retraining, Eclipse Retraining, Firefox with Group Centroids, Firefox No Retraining, Firefox Retraining, Open Office No Retraining and Open Office Retraining respectively.
Figure 33: MULAN Results with Firefox and No Retraining

Figure 34: MULAN Results with Firefox and Retraining
Figure 35: MULAN Results with Open Office and No Retraining

Figure 36: MULAN Results with Open Office and Retraining
In order to test the effectiveness of multilabel classification, we answer the following questions.

- Does multilabel classification outperform all singular similarity methods?
- Does the fusion rule impact the multilabel classifier performance?
- Does the training scheme impact the multilabel classifier performance?
- Does multilabel classification outperform the naive fusion scheme?
- Does increasing the list size improve the overall recall in duplicate classification?

7.9.1 Does multilabel classification outperform all singular similarity methods?

Multilabel classification when using the SUM rule provided improvements ranging from 7% to 41% in Eclipse, 8% to 39% in Firefox with Group Centroids, 10% to 44% in Firefox and 9% to 31% in Open Office. Multi-label classification outperforms the single classifier in all instances as shown in Tables 25, 26, 27 and 28.

7.9.2 Does the fusion rule impact the multilabel classifier performance?

Using the SUM rule as opposed to the MAX rule in the selection of the final list of candidate duplicate matches led to a better performance. One of the reasons is that due to the nature of the English language a high similarity score to an unrelated document could be offered by one of the suggested measures. Any errors caused by such anomalies are magnified when using the MAX rule. In contrast, the SUM rule aggregates the scores of various classifiers leading to a lower error rate. In this regard, we feel further investigation into fusion rules is necessary to further reduce the potential for false matches caused by the variance in expression in the English language.

7.9.3 Does the training scheme impact the multilabel classifier performance?

The non retraining methodology consistently provided better results. The reasoning is due to the variability in the data. When retraining, the local data exhibits noise
that is not captured in the training data preceding it. For example, in one round of retraining we may have more observations where all similarity methodologies can capture the problem report, but in the test set we may not have any samples that match the training data. Thus, the classifier cannot capture the subtle variability across windows of time. Training once when the project has started allows the classifier to have the capability to capture a broader range of samples.

7.9.4 Does multilabel classification outperform the naive fusion scheme?

For the Eclipse and Open Office datasets the multilabel approach performed as well as the naive approach. For the Firefox dataset, the multilabel approach performed 2% better. The multilabel approach is superior to the naive approach for several reasons, namely:

- **Speed**: the naive approach obtains top 20 lists from all 24 methods, the multilabel approach on average selects 12 out of the 24 methods.

- **Effectiveness**: the multilabel approach captures reports the naive fusion approach misses by considering only the most likely matches. The naive approach is confounded by multilabel methods generating spurious top-20 lists.

It is essential to understand the limitations of the multilabel approach, how they arise and how they can be mitigated. To understand where the multilabel approach struggles we begin by evaluating the performance of the multilabel approach as the number of detectable methods changes.
Figure 37: Impact on Recall as Number of Detectable Methods Changes

From Figure 37 we notice that as the number of methods, that can detect a duplicate, increases so does the overall recall. A positive correlation, of 0.61, 0.62 and 0.73 for Eclipse, Firefox and Open Office respectively, was found between the number of methods and the multilabel classifier’s performance. The multilabel method struggles when less than 20% of the methods can detect a duplicate. This is intuitive, however it does not explain why the classification fails when few methods are capable of detecting the duplicates. In order to understand this phenomenon we explore the proportion of methods in the training and testing set for all three projects.
Figure 38: Changes in Test Set to Training Set Ratio as Number of Detectable Methods Changes

From Figure 38 we notice that the training set does not contain sufficient samples when less than 20% of the methods can correctly classify a report. Each dataset plot in the figure represents the best MULAN approach for that particular dataset. In the figure, the x-axis represents the number of methods that can detect a duplicate and the y-axis represents the sample size in the test set compared to the training set. We notice that the training set is heavily biased towards a large number of methods being able to detect a duplicate. Thus, the classifier is biased towards learning these characteristics. This is further corroborated by a strong positive correlation in the test to train ratio and the recall for the specific ratio of methods. Both Firefox and Open Office had a positive correlation of 0.60 and 0.61 respectively. Eclipse, while having a correlation of 0.23 was heavily biased by the presence of outliers. A similar trend was also observed when using retraining. Thus, a better classification system that utilizes an ensemble of methods, such as random forests, can offer better results. We leave the analysis of other base classification systems as future work.

7.9.5 Does increasing the list size improve the overall recall in duplicate classification?

Table 31 shows the effect on the recall rate as the list size changes from the top-20 to the top-100 list. The results are provided for the best non retraining and retraining based methods. A list size of 20 is typically chosen as it is unlikely that a user
Table 31: Duplicate Classification: Effect of List Size on MULAN Recall

<table>
<thead>
<tr>
<th>List Size</th>
<th>No Retraining</th>
<th>Retraining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eclipse</td>
<td>Firefox</td>
</tr>
<tr>
<td>20</td>
<td>63.38%</td>
<td>68.09%</td>
</tr>
<tr>
<td>40</td>
<td>69.96%</td>
<td>74.81%</td>
</tr>
<tr>
<td>60</td>
<td>73%</td>
<td>78.48%</td>
</tr>
<tr>
<td>80</td>
<td>74.49%</td>
<td>80.85%</td>
</tr>
<tr>
<td>100</td>
<td>75.10%</td>
<td>82.35%</td>
</tr>
</tbody>
</table>

will explore more than 20 reports. However, as our framework has been built as a triaging aide, one can expect the triager to explore additional reports if it provides measurable improvements in performance. We notice that changing the list size from 20 to 40 increases the duplicate classification recall by an average of 6.5%. However, increasing the list size from 40 to 100, that is an additional 60 reports, only provides on average a 5.69% improvement in performance. Thus, if necessary the triage team may choose to change the list size to 40 to correctly classify more duplicates without incurring excessive overhead in reading through additional reports.

### 7.10 Effects of Data Set Size

Figures 39, 40, 41 and 42 shows the effect of increase in data set size on the recall rate. We notice that the recall rate for all three datasets stabilize over time. Thus, despite a growing dataset size, the multilabel approach does not appear to deteriorate. However, Open Office suffers a stronger decline in performance over time. As discussed throughout this thesis, Open Office has seen a steady decline in the number of users and usage since 2005 until the project was officially discontinued in 2011. Thus, one can expect that duplicate reports submitted since 2005 (past the 50% size of the dataset) may be more difficult to detect.
Figure 39: Impact on Recall Rate as Dataset Size Increases (Eclipse)

Figure 40: Impact on Recall Rate as Dataset Size Increases (Firefox with Group Centroids)
7.11 Threats to Validity

The similarity measures between the two reports selected in this work do not appear to be the best possible ones. Research in [11] indicated a potential recall rate of 84% using topic modeling for the Eclipse 2008 dataset. However, our objective was to demonstrate that multi-label classification can be used to boost the performance
of simple similarity metrics. The application of our duplicate classification approach to a three Bugzilla based repositories, no matter how large, can be perceived as a threat to validity. The approaches described can be easily transposed to any problem repository.

7.12 Conclusions

In this chapter we demonstrated the effectiveness of multilabel classification in enhancing the performance of individual problem report similarity measures. Our results indicate that a multi-label classification scheme can improve the base classifier up to 44%, while providing overall results that are comparable with the current corpus of research. The multi-label classifier does not degrade its performance significantly as the data set size increases.

Multi-label classification addresses one of the major issues faced by singular classifiers. Due to the depth and diversity of the human language a method that works on a particular data set or portion of the data may not work on another. We demonstrated this effect by investigating the effect of the problem report factors, duplicate group size and classification method. Little emphasis has been placed to date on the impact of time and the growth in repository size on duplicate report classification performance. Many research methods appear to suffer performance penalty as the data set size increases. The multi-label classification approach did not appear to suffer any major degradation in performance with time. As researchers we must develop systems that are robust over large volumes of data. Having systems that work in controlled environments but not on a realistic data set makes it too risky for practitioners to implement novel research ideas.

The multilabel approach provided similar results to the naive fusion approach for two of the datasets. The Firefox dataset outperformed the naive approach by 2%. One reasoning for this is the selection of the base classifier. The fact that the multilabel approach works as well as a naive approach indicates that the classifier was performing adequately. The challenges faced by the multilabel approach occur when only a few similarity measures are able to correctly classify a sample. In such instances, the multilabel approach fails to generate an adequate subset. Given that our approach tried only a simple decision tree, better results would be expected when using an ensemble approach, such as random forests. We leave this as part of our future work.
### Table 32: Fully Automated Triaging - Eclipse

<table>
<thead>
<tr>
<th>Raw Data 85% Threshold</th>
<th>Actual</th>
<th>Correctly Predicted</th>
<th>Recall</th>
<th>Top-20 Recall Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>302566</td>
<td>226023</td>
<td>74%</td>
<td>NA</td>
</tr>
<tr>
<td>Duplicate</td>
<td>39209</td>
<td>25200</td>
<td>64%</td>
<td>64%</td>
</tr>
</tbody>
</table>

**8 A Fully Automated Framework for Classifying Problem Reports**

From Section 4 we can develop a methodology to determine whether each incoming report is primary or duplicate. Once we have determined that the report is in fact duplicate, we can apply the multi-label classification technique from Section 7 to select the 20 most likely matches. In this section we present the results for the fully automated problem report classification system by utilizing the Eclipse, Firefox and Open Office datasets.

#### 8.1 Eclipse

We select the raw dataset and a retraining size of 20,000 with 12 random features and no feature selection. We employ the 85% voting threshold to distinguish between primary and duplicate reports. This threshold provides a TPR of 75% and TNR of 64%, thus 25% of all primary reports will be classified as duplicate and 36% of all duplicate reports will be classified as primary. The predicted duplicate reports are then passed to the MULAN multi-label classification system to determine the top 20 list of most likely matches.

Given that duplicate classification is the second stage in the triaging process, we present results using no prior assumptions on the data, and using the assumption that the triager has removed problematic reports.

Table 32 shows the results of the fully automated process. When using no triager intervention, we are able to correctly categorize 64% of the correctly classified duplicate problem reports in the top 20 list.

Incorrectly classified primary reports will fail to generate any plausible matches in a top 20 list. Thus, the trigger will assign such reports to the developer to develop a fix. The bigger workload challenge are the incorrectly classified duplicate, as they will also get assigned to a developer. In this case the developer must look at 23081 (i.e. 39209-25200 + 0.36*25200) reports as opposed to 363770 reports. In
other words, we provide nearly a 94% reduction in triager workload. Unlike related research which assumes the duplicate report is already known, our approach uses the fully automated system to assist the triager.

### 8.2 Firefox

We select the raw dataset and a retraining size of 10,000 with 20 random features and no feature selection. We employ the 75% voting threshold to distinguish between primary and duplicate reports. This threshold provides a TPR of 68% and TNR of 61%, thus 32% of all primary reports will be classified as duplicate and 39% of all duplicate reports will be classified as primary. The predicted duplicate reports are then passed to the MULAN multi-label classification system to determine the top 20 list of most likely matches.

Given the challenges faced by the presence of poorly marked reports, we repeat the procedure by also using the clean dataset. For the clean dataset we select a retraining size of 10,000 with 20 random features and no feature selection. We employ the 60% voting threshold to distinguish between primary and duplicate reports. This threshold provides a TPR of 77% and TNR of 68%, thus 23% of all primary reports will be classified as duplicate and 32% of all duplicate reports will be classified as primary. The predicted duplicate reports are then passed to the MULAN multi-label classification system to determine the top 20 list of most likely matches.

Given that duplicate classification is the second stage in the triaging process, we present results using no prior assumptions on the data, and using the assumption that the triager has removed problematic reports.

Table 33 shows the results of the fully automated process. When using no triager intervention, we are able to correctly categorize 74% of the correctly classified dupli-

<table>
<thead>
<tr>
<th>Raw Data 75% Threshold</th>
<th>Actual</th>
<th>Correctly Predicted</th>
<th>Recall</th>
<th>Top-20 Recall</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>79048</td>
<td>53454</td>
<td>68%</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Duplicate</td>
<td>22157</td>
<td>13456</td>
<td>61%</td>
<td>74%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clean Data 60% Threshold</th>
<th>Actual</th>
<th>Correctly Predicted</th>
<th>Recall</th>
<th>Top-20 Recall</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>28813</td>
<td>22054</td>
<td>77%</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Duplicate</td>
<td>13323</td>
<td>9004</td>
<td>68%</td>
<td>70%</td>
<td></td>
</tr>
</tbody>
</table>

Table 33: Fully Automated Triaging - Firefox
cate problem reports in the top 20 list. When using triager assistance in removing problematic reports, we are able to correctly categorize 70% of the correctly classified duplicate reports in a top 20 list.

Incorrectly classified primary reports will fail to generate any plausible matches in a top 20 list. Thus, the trigger will assign such reports to the developer to develop a fix. The bigger workload challenge are the incorrectly classified duplicate, as they will also get assigned to a developer. In this case the developer must look at 12000 (i.e. 22157-13456 + 0.26*13456) reports as opposed to 111,205 reports. In other words, we provide nearly a 90% reduction in triager workload. Unlike related research which assumes the duplicate report is already known, our approach uses the fully automated system to assist the triager.

8.3 Open Office

We select the raw dataset and a retraining size of 10,000 with 20 random features and no feature selection. We employ the 80% voting threshold to distinguish between primary and duplicate reports. This threshold provides a TPR of 78% and TNR of 44%, thus 22% of all primary reports will be classified as duplicate and 56% of all duplicate reports will be classified as primary. The predicted duplicate reports are then passed to the MULAN multi-label classification system to determine the top 20 list of most likely matches.

Given the challenges faced by the presence of poorly marked reports. We repeat the procedure by also using the clean dataset. For the clean dataset we select a retraining size of 10,000 with 20 random features and no feature selection. We employ the 75% voting threshold to distinguish between primary and duplicate reports. This threshold provides a TPR of 79% and TNR of 54%, thus 21% of all primary reports will be classified as duplicate and 46% of all duplicate reports will be classified as primary. The predicted duplicate reports are then passed to the MULAN multi-label classification system to determine the top 20 list of most likely matches.

Given that duplicate classification is the second stage in the triaging process, we present results using no prior assumptions on the data, and using the assumption that the triager has removed problematic reports.

Table 34 shows the results of the fully automated process. When using no triager intervention, we are able to correctly categorize 66% of the correctly classified duplicate problem reports in the top 20 list. When using triager assistance in removing problematic reports, we are able to correctly categorize 63% of the correctly classified problem reports in the top 20 list.
Table 34: Fully Automated Triaging - Open Office

duplicate reports in a top 20 list.

Incorrectly classified primary reports will fail to generate any plausible matches in a top 20 list. Thus, the trigger will assign such reports to the developer to develop a fix. The bigger workload challenge are the incorrectly classified duplicate, as they will also get assigned to a developer. In this case the developer must look at 12196 (i.e. 17118-7457 + 0.34*7457) reports as opposed to 117,195 reports. In other words, we provide nearly a 90% reduction in triager workload. Unlike related research which assumes the duplicate report is already known, our approach uses the fully automated system to assist the triager.
9 Conclusions and Future Work

The work presented here represents the first effort at building an automated framework for problem report classification. While the related literature showed a few efforts at building such a system, the results were either poor or the framework did not adequately model a real environment. Our approach makes no assumptions on the prior state of the repository, and classifies reports based purely on the merit of the similarity score statistics, document attributes and user behavior. The proposed framework was tested on three large-scale open source repositories, namely Eclipse, Open Office and Firefox. The research results indicate that the chosen metrics can allow us to differentiate between new and existing reports. For existing reports, we showed that a single silver bullet approach does not exist and instead the intelligent fusion of approaches allows us to utilize the benefits of each approach. Our experimental analysis was conducted on the complete datasets from Eclipse, Firefox and Open Office. Tackling such large datasets is a daunting process, and we quantified the cost and storage requirements for analysis.

One of the assumptions made in our study was the manual removal of problematic reports. The removal of such reports demonstrated a substantial improvement in performance in the Firefox and Open Office datasets. The reason for this improvement was due to the likelihood that the reports were incorrectly labeled by the triagers. In future, the selection of problematic reports should be automated in order to ensure the triager does not need to be involved in the triaging process, and only acts as an aide to the system.

Our approach heavily relied on non semantic based approaches from natural language processing. Given that the English language contains noun and verb structure, we propose to apply parts of speech tagging techniques to the base similarity measures. However, the application of parts of speech tagging cannot be performed directly as taggers are trained based on standard dictionaries. The dictionary of words found in problem repositories contains both English and non English (technical) terms. One approach to solve this problem is to develop the dictionary by evaluating requirement specification documents and interviewing key agents in each project.

The research presented here began by introducing the concept of human vs. machine language. We theorized that machine language is vastly different from human language, and the that while techniques have been created in the domain of natu-
ral language processing and information retrieval, the marriage of the two domains remains relatively unexplored. Topic modeling, parts of speech tagging and sentiment analysis are many techniques currently being applied in the natural language processing domain on structured documents with proper syntax and grammatical considerations. Problem reports represent a domain where such structure does not exist, and direct application of these techniques may not be possible. We encourage future researchers to begin by asking the question: "What exists at the intersection of human and machine language?" Only after we have adequately defined the semantics of that language can we begin developing methods that can address the challenges faced in large-scale, unstructured datasets. The blind, black box, application of machine learning and information retrieval techniques will only create results that do not scale to the real world.
References


[34] Bugzilla Fields: https://bugzilla.mozilla.org/page.cgi?id=elds.html - retrieved October 2, 2013


