Performance of IR Models on Duplicate Bug Report Detection: A Comparative Study

by

Nilam Kaushik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

Abstract

Open source projects incorporate bug triagers to help with the task of bug report assignment to developers. One of the tasks of a triager is to identify whether an incoming bug report is a duplicate of a pre-existing report. In order to detect duplicate bug reports, a triager either relies on his memory and experience or on the search capabilties of the bug repository. Both these approaches can be time consuming for the triager and may also lead to the misidentification of duplicates. It has also been suggested that duplicate bug reports are not necessarily harmful, instead they can complement each other to provide additional information for developers to investigate the defect at hand. This motivates the need for automated or semi-automated techniques for duplicate bug detection.

In the literature, two main approaches have been proposed to solve this problem. The first approach is to prevent duplicate reports from reaching developers by automatically filtering them while the second approach deals with offering the triager a list of top-N similar bug reports, allowing the triager to compare the incoming bug report with the ones provided in the list. Previous works have tried to enhance the quality of the suggested lists, but the approaches either suffered a poor Recall Rate or they incurred additional runtime overhead, making the deployment of a retrieval system impractical. To the extent of our knowledge, there has been little work done to do an exhaustive comparison of the performance of different Information Retrieval Models (especially using more recent techniques such as topic modeling) on this problem and understanding the effectiveness of different heuristics across various application domains.

In this thesis, we compare the performance of word based models (derivatives of the Vector Space Model) such as TF-IDF, Log-Entropy with that of topic based models such as Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA) and Random Indexing (RI). We leverage heuristics that incorporate exception stack frames, surface features, summary and long description from the free-form text in the bug report. We perform experiments on subsets of bug reports from Eclipse and Firefox and achieve a recall rate of 60% and 58% respectively. We find that word based models, in particular a Log-Entropy based weighting scheme, outperform topic based ones such as LSI and LDA.

Using historical bug data from Eclipse and NetBeans, we determine the optimal time frame for a desired level of duplicate bug report coverage. We realize an Online Duplicate Detection Framework that uses a sliding window of a constant time frame as a first step towards simulating incoming bug reports and recommending duplicates to the end user.

Acknowledgements

I would like to thank my supervisor, Dr. Ladan Tahvildari for her constant support and feedback throughout my research. I am glad to have joined a thesis-based Masters under her supervision, it has truly changed my perspective on academia. It has been a pleasure working with Ladan, who was always there to help, not only in research related matters but also as an older sister.

I would also like to thank Dr. Kostas Kontogiannis and Dr. Lin Tan for spending the time to read my thesis and for providing their feedback. I would like to acknowledge Dr. Giuliano Antoniol for his guidance which eventually steered me towards this research direction.

I also take this opportunity to express my profound gratitude to my parents, Mohinder and Sunita Kaushik, for giving me the freedom to explore my interests and for being so patient. To my brother, Pashupati, who has his own adorable ways of expressing his love for me. I also wish to thank members of my extended family for always encouraging me and for being proud of my accomplishments. Lastly, I am grateful to my best friends for being with me through thick and thin - each one of you is special.

Dedication

To my best friend

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large open source software development projects incorporate bug tracking systems that enable users and developers to track issues, including bugs, feature requests, enhancements, organizational issues and refactoring activities. Maintenance activities account for over two-thirds of the life cycle cost of a software system [7], with billions of dollars lost due to software defects [48]. As a result, many software projects rely on bug tracking systems to manage corrective maintenance activities [18].

Bug tracking systems have several advantages. By allowing the users of the software to be the testers of the software, they increase the possibility of revealing bugs, thereby improving software quality [27]. This facilitates tracing the evolution of the software project by tracking the number of outstanding issues [9, 44], and it also provides a means for geographically distributed developers and users to communicate about aspects of project development [27, 41]. In effect, a bug management system is a hub for users, developers and other stakeholders to engage in the development and maintenance aspects related to the project. Bug management is similar to task management in a service organization. Issues in bug tracking systems are tracked through bug reports, which are detailed natural language descriptions of defects, or problems in a software product. In open source software projects, bug reports are filed by users or developers who discover defects during their usage of the product. A bug report is a "relay" intended to solve a task. Different actors contribute to solving the task, and the information management system is the central node that dispatches subtasks to actors [39].

Though the benefits of bug tracking systems in software development projects are obvious, there are problems that arise as a consequence of their use. Some of the problems that may appear as a result of the usage of bug tracking systems are dynamic assignment of bug reports, change impact analysis and effort estimation, quality of bug report descriptions, software evolution and traceability and duplicate bug report detection.

Due to the nature of open source software development, projects rely on a large number of users to be involved in the testing activities, which are asynchronous. As bug reporting is asynchronous and uncoordinated, the same bugs may be reported by different users over time. The cost of users searching the bug tracking system to determine whether their problem has already been reported is higher than the cost of creating a bug report [53]. Due to the large volume of incoming bug reports, open source projects hire triagers to assist with the task of bug report assignment to developers. One of the tasks of a triager is to identify whether an incoming bug report is a duplicate of a pre-existing report. In order to detect duplicate bug reports, a triager either relies on his memory and experience or on the search capabilties of the bug repository. Both these approaches can be time consuming for the triager and may also lead to the misidentification of duplicates. Furthermore, in a recent survey by Just et al. [19], it was found that the identification of duplicate bug reports was one of the wanted features for next generation bug tracking systems. Therefore, there is a need for automated or semi-automated techniques to assist triagers with the process of identifying/filtering duplicate bug reports. In this thesis, we focus on the aspect of duplicate bug report detection.

1.1 Motivation

In this section, we highlight the problem of bug report duplication and its impact on software development and emphasize the need for a system to detect duplicate bug reports. In their characterization study on bug repositories and the search and analysis of bug reports, Cavalcanti et al. [10] investigated 8 open source projects. Their findings are summarized in Table 1.1 , these statistics were collected on bug reports from the beginning of each individual project till the end of June 2008.

Project	**Domain**	**Code size**	**Bugs**	**Life-time (yrs)**	**% Bug report duplication**
Bugzilla	Bug tracker	55K	12829	14	68
Eclipse	IDE	6.5M	130095	7	19
Epiphany	Browser	100K	10683	6	32
Evolution	E-mail client	1M	72646	11	43
Firefox	Browser	80K	60233	9	38
GCC	Compiler	4.2M	35797	9	18
Thunderbird	E-mail client	310K	19204	8	49
Tomcat	Application server	200K	8293	8	8

Table 1.1: Project Characteristics

The average time spent on searching and analysing bug reports before opening a new bug report is 12.5 minutes. They also approximate an average of 48-person hours spent per day only on searching similar bug reports. The highest amount of bug duplication is noticed in Bugzilla at 68% followed by Thunderbird which has 49% duplication.

Furthermore, some projects have special guidelines and websites dedicated to information about duplicate bug reports. For example, the Mozilla project maintains a "Most Frequently reported bugs" page [8], to track the bugs that are reported most frequently and counts the number of duplicate instances of the bugs. The web page states that this is done in order to minimize the amount of duplicate bugs entered into Bugzilla, which in turn saves QA engineers time to triage bugs. It is evident from the reported statistics that bug report duplication is a common problem in open source software projects.

1.1.1 Motivating Examples

We present two duplicate bug report pairs from the Eclipse project to show the potential of using IR methodologies to detect duplicate bug reports.

Bug 323444 and Bug 330258

In Eclipse, Bug 323444 and 330258 are about a ConcurrentModificationException in the source viewer. Table 1.2 presents the information about the summary and long description of the individual bugs.

BugID	Summary	Long Description
323444	[Undo][Commands] java.util. ConcurrentModificationException when trying to get the undo history from a source viewer	I got the following exception once when attempting to get the undo history from a source viewer in my code. Looking at the implementation of this method, I cannot seem to see how this could occur since all access to this list in DefaultOperationHistory are synchronized. Maybe it is the creation of the iterator outside of the synchronized block? Note that I have been unable to reproduce this since I first saw it since it seems to be a rare race condition. Exception Stack Trace: java.util.ConcurrentModificationException at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372) at java.util.AbstractList$Itr.next(AbstractList.java:343) at org.eclipse.core.commands.operations. DefaultOperationHistory.filter(DefaultOperationHistory.java:558) at org.eclipse.core.commands.operations. DefaultOperationHistory.getUndoHistory(DefaultOperationHistory.java:843)
330258	[Undo] ConcurrentModificationException in DefaultOperationHistory.filter()	The iterator used in DefaultOperationHistory.filter(...) is not properly synchronized against comodification, allowing asynchronous modification of the list between obtaining the iterator and entering the following synchronized block. This leads to java.util.ConcurrentModificationException at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372) at java.util.AbstractList$Itr.next(AbstractList.java:343) at org.eclipse.core.commands.operations.DefaultOperationHistory.filter (DefaultOperationHistory.java:558) at org.eclipse.core.commands.operations.DefaultOperationHistory. getUndoHistory(DefaultOperationHistory.java:843)

Table 1.2: Duplicate Bug Report Pair-I

We can gain insight from this example. It is easy to notice that some common terms such as “Undo” and “ConcurrentModificationException” are shared between the two reports. While the bug report summaries do not necessarily suggest that the two reports are duplicates, the stack frames from the exception traces in the long descriptions of both reports strongly indicate that the reports are duplicates of each other. Also, surface features such as the product and component are common to both reports.

Bug 348680 and Bug 264190

Bugs 348680 and Bug 264190 in Eclipse relate to the inability to resize a status bar. The summaries and long descriptions of the individual bugs are presented in Table 1.3. Unlike the previous example, the bug report summaries in this case strongly indicate that the two reports are duplicates. However, there is also a lot of extraneous information in the long description of the bug reports that can act as noise when computing similarity. In this example, however, due to the nature of the problem, there is no exception stack trace in the long description. Terms such as status, bar, line are common to both reports and the bug reports also share the same surface features of Product and Component.

BugID	**Summary**	**Long Description**
348680	The status bar is not resizable	The status bar(like the one that shows line:column number when the java editor is open) should be resizable. Sometimes this if other items are added to the status bar , for eg by adding view as fast view icons , then the status bar showing the line:column detail becomes shortened, and there is no way to expand it(i.e the width). I have attached a screen shot for more details. Thanks Reproducible: Always Steps to Reproduce: 1.Open Java Editor, see that the status bar showing LINE:COL is visible 2.In the status bar , keep adding some views as fast-view (show view as fast view) 3. the Line:col status bar would start to contract.
330258	[Trim] Status bar element with the line & column number can get cut off with small window sizes	When you resize the workbench window it is possible to get into a state where the line number widget is not shown just before wrapping. This is an issue for screen readers as they walk the visible widgets and then report on thier contents. As the line number is an important piece of information for people who cannot navigate visually it can make the editors much harder to use. There is no redraw function such as reset perspective to correct this as far as I could find.

Table 1.3: Duplicate Bug Report Pair-II

In this work, we use different combinations of summary, description, exception stack frames(wherever available) and surface features such as Product, Component and Classification information and observe their the effect on retrieval performance.

1.2 The Problem

Due to the nature of open source software development, projects rely on a large number of users to be involved in the testing activities, which are asynchronous. As bug reporting is asynchronous and uncoordinated, the same bugs may be reported by different users over time. The cost of users searching the bug tracking system to determine whether their problem has already been reported is higher than the cost of creating a bug report [53]. Due to the large volume of incoming bug reports, open source projects hire triagers to assist with the task of bug report assignment to developers. One of the tasks of a triager is to identify whether an incoming bug report is a duplicate of a pre-existing report. In order to detect duplicate bug reports, a triager either relies on his memory and experience or on the search capabilties of the bug repository. Both these approaches can be time consuming for the triager and may also lead to the misidentification of duplicates.

Some techniques have been proposed to help triagers with the process of duplicate bug detection. These can be broadly categorized into two approaches. The first approach is to prevent duplicate reports from reaching developers by automatically filtering them [18]. The second approach deals with offering the triager a list of top-N similar bug reports, allowing the triager to compare the incoming bug report with the ones provided in the list [39, 45, 53, 51]. If the triager finds a report in the list that matches the incoming bug report, it is marked as a DUPLICATE of the existing report. In such an event, the existing bug report is considered as the master report.

It has been suggested that duplicate bug reports are not necessarily harmful, instead they can complement each other to provide additional information for developers to investigate the defect at hand [5]. This motivates the need for duplicate bug detection as its benefits are two fold- first, it saves triagers' time to find related bug reports and secondly, it helps developers gather more information about the defect at hand, in turn reducing the time to resolve a bug. Previous works have tried to enhance the quality of suggested lists using Information Retrieval (IR) techniques, but the approaches either suffered a poor Recall rate [39, 18] or they incurred additional runtime overhead [53, 45], making the deployment of a retrieval system impractical. To the extent of our knowledge, there has been little work done to do an exhaustive comparison of the performance of different IR models (especially using more recent techniques in IR such as topic modeling) on this problem and understanding the effectiveness of different heuristics across various application domains.

1.3 Thesis Contributions

The contributions of the work are summarised as follows:

- Conduct a comprehensive survey on the usage of Information Retrieval techniques in the area of duplicate bug report detection.
- Evaluate the impact of using different heuristics in helping duplicate bug report detection across different application domains.

- For a required coverage of duplicate detection, determine the optimal time frame for its query space based on the project's historical data.
- Realize an online duplicate bug report detection framework that uses a sliding window of a constant time frame to maximise the potential for capturing duplicate bug reports. It is the first step towards simulating incoming bug reports and recommending duplicates.

1.4 Thesis Organization

The disseration is organized as follows: Chapter 2 provides the background concepts and review of the research related to this thesis. It describes the structure and life-cycle of a bug report followed by an overview of IR concepts. Chapter 3 surveys the related work in the area of duplicate bug report detection. Chapter 4 presents an overview of the proposed techniques. Chapter 5 describes the experimental setup and data sets used for the experimental studies. It also reports on the results of using the proposed approach on the experimental studies. Chapter 6 draws conclusions from the presented research and highlights the research contributions. It also outlines potential future directions that could be pursued from this research.

Chapter 2

Background Concepts

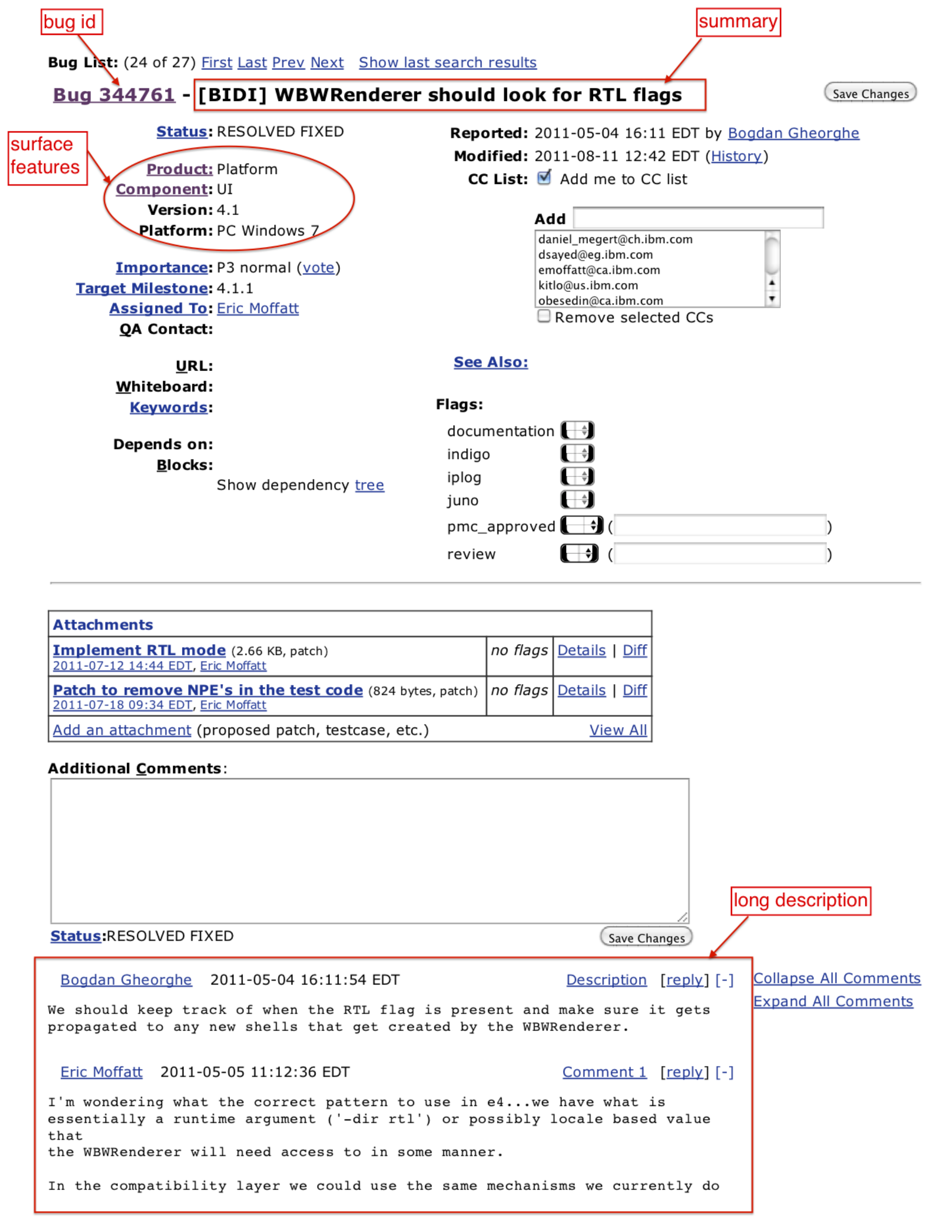
This chapter begins by providing some background on the structure and life-cycle of a bug report. This is followed by an overview of Information Retrieval techniques and processing stages in Natural Language Processing. The chapter also provides an in-depth discussion on Information Retrieval models, both word and topic based ones. Finally, the chapter ends by surveying the applications of Information Retrieval in the field of Software Engineering.

2.1 Bug Reports

Large-scale open source projects such as Eclipse and Mozilla use Bugzilla as their bug tracking management system. Issues in bug tracking systems are tracked through bug reports. Bug reports are detailed natural language descriptions of issues. In this section, we provide an overview of the bug report- its structure and its life-cycle in order to better understand the types of data that can be extracted from bug reports.

2.1.1 Structure of a Bug Report

Figure 2.1 is an example of a bug report from the Eclipse project. There are certain fields such as the bug report Creation date, Reporter name, Bug id that are auto-populated when a user creates a report. Other fields such as the report Summary, Product, Component, Hardware, OS, Version, Priority, Severity, Description etc are populated by the user and can change over the life-cycle of the bug, as described in Section 2.1.2. Reporters can also provide attachments containing information such as error logs and screenshots to highlight the problem at hand. The intent of adding such information is to assist the developer in the process of bug resolution.



Bug List: (24 of 27) First Last Prev Next Show last search results

Bug 344761 - **[BIDI] WBWRenderer should look for RTL flags**

Save Changes

Status: RESOLVED FIXED

Product: Platform
Component: UI
Version: 4.1
Platform: PC Windows 7

Importance: P3 normal (vote)
Target Milestone: 4.1.1
Assigned To: Eric Moffatt
QA Contact:

URL:
Whiteboard:
Keywords:

Depends on:
Blocks:
Show dependency tree

Reported: 2011-05-04 16:11 EDT by Bogdan Gheorghe
Modified: 2011-08-11 12:42 EDT (History)
CC List: ✓ Add me to CC list

Add
daniel_megert@ch.ibm.com
dsayed@eg.ibm.com
emoffatt@ca.ibm.com
kitlo@us.ibm.com
obesedin@ca.ibm.com
Remove selected CCs

See Also:

Flags:
documentation
indigo
iplog
juno
pmc_approved ( )
review ( )

Attachments		
Implement RTL mode (2.66 KB, patch) 2011-07-12 14:44 EDT, Eric Moffatt	*no flags*	Details | Diff
Patch to remove NPE's in the test code (824 bytes, patch) 2011-07-18 09:34 EDT, Eric Moffatt	*no flags*	Details | Diff
Add an attachment (proposed patch, testcase, etc.)		View All

Additional Comments:

Status:RESOLVED FIXED

Save Changes

Bogdan Gheorghe 2011-05-04 16:11:54 EDT Description [reply] [-]

Collapse All Comments
Expand All Comments

```
We should keep track of when the RTL flag is present and make sure it gets
propagated to any new shells that get created by the WBWRenderer.
```

Eric Moffatt 2011-05-05 11:12:36 EDT Comment 1 [reply] [-]

```
I'm wondering what the correct pattern to use in e4...we have what is
essentially a runtime argument ('-dir rtl') or possibly locale based value
that
the WBWRenderer will need access to in some manner.

In the compatibility layer we could use the same mechanisms we currently do
```

Figure 2.1: A Sample Bug Report from Eclipse

The “Summary” field is a short, concise description of the bug that is populated by the reporter. In the example in Figure 2.1, this field reads “[BIDI WBWBRenderer should look for RTL flags”. The full-description of the report, in the “Description” field, contains additional information from the reporter describing the problem. Other developers can later on provide additional comments and information pertaining to the bug. This gets appended in the “Description” field as “Additional comments”.

The unstructured text in a bug report is comprised of the summary of the bug report as well as the description of the bug (including the additional comments). Other categorical information such as Component, Product, OS etc. can also be useful in addition to the summary for the bug duplicate detection problem. These are referred to as “Surface Features”. Other bug tracking systems such as JIRA[4] and TRAC [43] also have a similar structure for bug reports.

2.1.2 Life-cycle of a Bug Report

Bugs move through a series of states over their lifetime. The various states of a bug report in Bugzilla are captured in Figure 2.2. Depending on the project’s development process, states may be added or removed to suit the needs of the project.

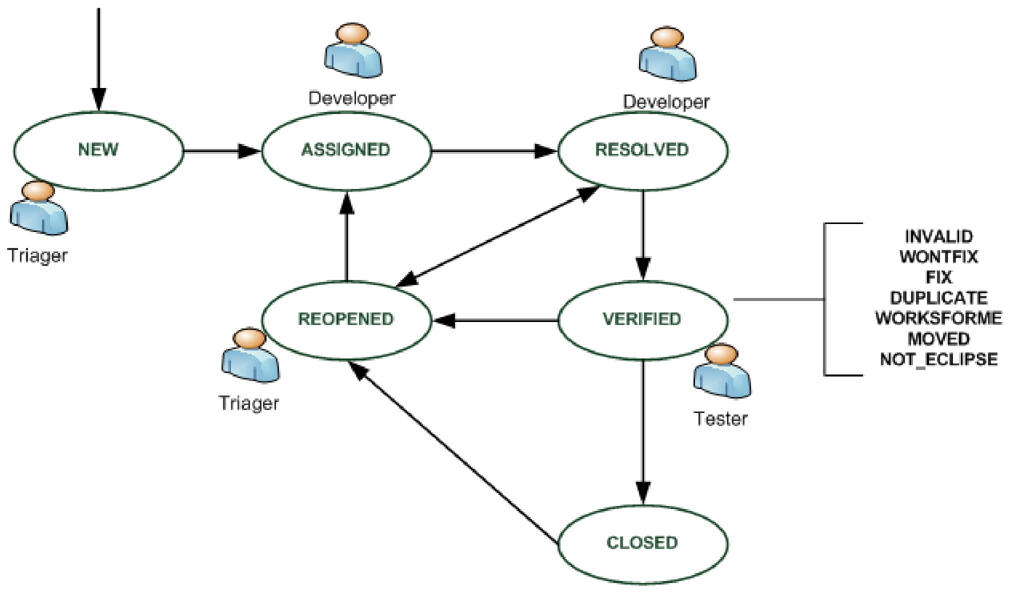


Figure 2.2: Different Possible States of a Bug Report During its Life-cycle

Resolution type	**Description**
INVALID	Indicates that the resolver deemed the report invalid
WONTFIX	Indicates that the resolver determined the bug will not be fixed
FIX	Indicates that a bug has been fixed
DUPLICATE	Indicates that this is a duplicate of another bug
WORKSFORME	Indicates that the resolver could not reproduce the problem
MOVED	Indicates that this bug report has been moved to another area
NOT_ECLIPSE	Indicates that this bug does not pertain to the Eclipse project

Table 2.1: Bug Resolution Types

We discuss the life-cycle of a bug report in Eclipse, which is similar to that of other projects. In Eclipse, when a bug report is submitted, its status gets set to NEW. Once the report is assigned to a developer for further investigation, its status is changed to ASSIGNED. Finally, when a report is resolved, its status is changed to RESOLVED. Depending on the nature of the bug, there are many possible resolutions of a bug. These are summarized in Table 2.1 . If there are no further issues, the bug report is marked CLOSED. If for some reason a previously resolved bug report gets reopened later on, it gets a status of REOPENED.

2.2 Information Retrieval Techniques

Information Retrieval (IR) is a discipline that deals with the retrieval of unstructured data, especially text documents, in response to a query, which may or may not be structured itself [16]. The goal of Information Retrieval is to develop models and techniques for retrieving information expressed in Natural language. The most common application of IR techniques today is in search engines. Natural Language Processing (NLP) is a field within computer science and linguistics that deals with the interaction between human languages and computers. Manning and Schutze [23] outline the following processing stages in NLP:

- **Tokenization**: is the process of turning a stream of text into words, phrases or other elements called tokens. It is done by removing capitals, punctuations and other special symbols or characters. A token is a word (a sequence of alphanumeric characters) surrounded by whitespace. There are various tokenizers, such as sentence tokenizers, word tokenizers and paragraph tokenizers etc.

- **Stemming**: is the process of reducing words to their ground form, or stem. For example, the words "distributed" and "distributing" are reduced to "distribute", the root word. Verbs are also reduced to their root. The Porter Stemmer algorithm [34] is a de-facto algorithm used in English stemming.

- **Stopword Removal**: involves removing prepositions, articles, pronouns and conjunctions such as "a", "the", "that", "of", "this" etc. These words do not carry important information but are present to make meaningful sentences. Stopwords may distort similarity calculations and are typically removed. Stopwords can be accumlated in the form of a stopword list, the goal being that any words in a corpus that occur in the stopword list are explicitly removed. As stopword lists can vary depending on the nature of the data[16], there is no universal stopword list. However, the SMART stopword list [21] has been used commonly in NLP research.

- **Vector Space Model**: After tokenization, stemming and stop word removal, each document in the corpus is represented as a "bag of words", in which the ordering of words is ignored. For example, the sentences "John is quicker than Mary" and "Mary is quicker than John" have the same vector representation. A document d_i is represented by the vector: $d_i = (w_{1i}, w_{2i}, ...)$ where w_{ij} is represents the number of times the term w_1 appears in document d_i. The terms in a vector can be weighted in several ways. Figure 2.4 shows an example with three document vectors in the vector space. After representing every document in the collection as a bag of words, the documents can be represented in a term-document matrix as shown in Figure 2.3.

- **Similarity Measures**: The most common measures of similarity used in Information Retrieval are Cosine, Dice and Jaccard [23].

  - The Jaccard coefficient measures the similarity between sets and is defined by the size of the intersection of two sets divided by the size of the union of sample sets, given as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

The Jaccard distance measures the dissimilarity between sets, and is given by:

$$1 - J(A, B) \tag{2.2}$$

– Dice coefficient is similar to the Jaccard coefficient but is gives twice the weight to the intersection of the sets:

$$s = \frac{2 \cdot |A \cap B|}{|A| + |B|} \tag{2.3}$$

– Cosine similarity is a measure of the similarity between two vectors. It measures the cosine of the angle between the vectors and is derived using the Euclidean dot product formula as follows:

$$cos\theta = \frac{A \cdot B}{||A||||B||} \tag{2.4}$$

The cosine of the angle between two vectors helps determine if the vectors point in the same direction or not, thus giving a sense of how similar the vectors are.

2.2.1 Measures of Accuracy

The performance of different IR models is measured using two fundamental metrics: Precision and Recall. Precision is the ratio of the number of relevant documents retrieved to the total number of documents retrieved. Meanwhile, Recall is the ratio of the number of relevant documents retrieved to the total number of relevant documents for a given query.

$$precision_i = \frac{correct_i \cap retrieved_i}{correct_i} \tag{2.5}$$

$$recall_i = \frac{correct_i \cap retrieved_i}{retrieved_i} \tag{2.6}$$

In using Precision and Recall, documents can be classified in four categories as outlined in Table 2.2

Precision and Recall are inversely related, in other words, as Recall goes up, Precision decreases and vice versa. It is easy to achieve 100% Recall by returning all the documents irrespective of their relevance to a query. Therefore, Recall alone is not enough to measure accuracy. Precision vs Recall graphs show the trade-off between the two measures. Both

Relevant	**Irrelevant**
Retrieved & Relevant	Retrieved & Irrelevant
Not Retrieved & Relevant	Not Retrieved & Irrelevant

Table 2.2: Document Classification

these metrics have a value between [0,1]. F-measure, the harmonic mean of precision and Recall is also used and is given by:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.7}$$

F-measure takes into account the trade-off between precision and Recall.

2.3 IR Models

In this section, we briefly describe two types of IR models- word based, which assume term independence, and topic based models which assume some latent structure in the usage of terms across documents.

2.3.1 Word based Models

In the motivating examples in Section 1.1, we observed the overlapping usage of common words in the duplicate bug reports. It is common to find domain-specific keywords in the bug reports descriptions and summaries since reporters often describe problems using words pertaining to the specific domain. These keywords can be identifiers such as class names, method names or terminology used to descibe specific domain concepts. We exploit the occurrence of such keywords in bug reports to infer duplicate bug report relationships with the help of word based IR models. The underlying model for the word based models is the Vector space model. In the Vector space model, each document is represented as a vector of terms. A collection of d documents with t terms is represented as a t x d term-document matrix shown in Figure 2.3 . The columns of the matrix are terms while the rows are the documents in the collection. Each element of the matrix represents the corresponding weight of the term in a document. For example, the weight of term T1 in document D1 is w_{11} and the weight of term T1 in document D2 is w_{12} and so on. A weight of zero means that the term does not exist in the document.

$$\left(\begin{array}{ccccc} & T_1 & T_2 & \cdots & T_t \\ D_1 & w_{11} & w_{21} & \cdots & w_{t1} \\ D_2 & w_{12} & w_{22} & \cdots & w_{t2} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ D_n & w_{1n} & w_{2n} & \cdots & w_{tn} \end{array}\right)$$

Figure 2.3: A Term-Document matrix

Consider documents D1, D2 and a query Q where D1 is represented as

$$D1 = 2T_1 + 3T_2 + 5T_3 \tag{2.8}$$

and D2 is represented as

$$D2 = 3T_1 + 7T_2 + T_3 \tag{2.9}$$

and the query Q is represented as

$$Q = 0T_1 + 0T_2 + 2T_3 \tag{2.10}$$

All these three vectors can be mapped into the vector space as shown in Figure 2.4. The similarity between the query Q and documents D1 and D2 can now be measured using the cosine similarity measure as follows:

$$cos\theta = \frac{A \cdot B}{||A||||B||} \tag{2.11}$$

It is obvious from the example, that between documents D1 and D2, D1 is more similar to query Q due to its proximity to Q in the vector space.

Terms in a term-document matrix can be represented by different weighting schemes. The weight of a term i in a document j can be given by $L(i,j) \cdot G(i)$, where L(i,j) is the local weight and G(i) is the global weight of term i. The local weight represents the weight of a term within a particular document. The global weight [13] is the value of the term i for the entire collection of documents. Two common local weights used are the

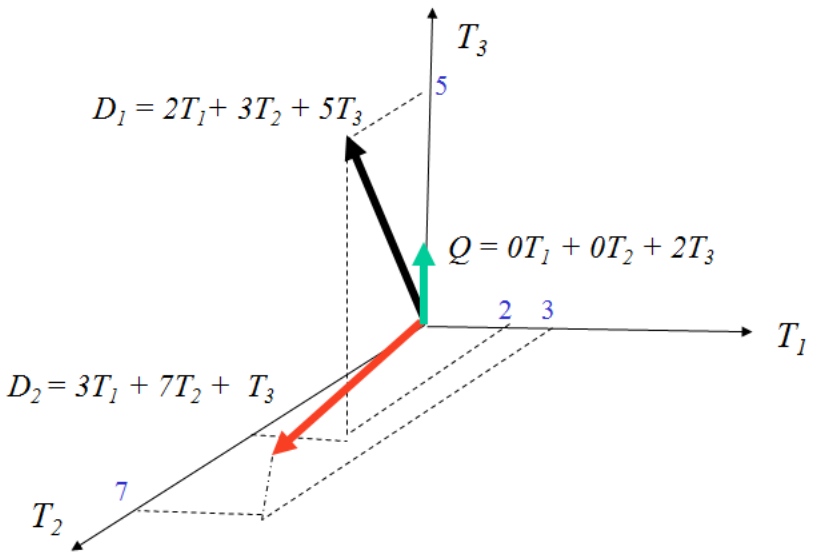


Figure 2.4: Documents and Query in the Vector Space

Term Frequency and a logarithmic function of the term frequency. Two common global weights are Entropy and Inverse Document Frequency. These are used in the following combinations:

Term Frequency - Inverse Document Frequency (TF-IDF)

In the TF-IDF model, the local weight is the term frequency, tf(i,j) and the global weight, idf(i) is the Inverse Document Frequency, given by equation 2.12.

$$log_2(\frac{N_{docs}}{N_i}) \tag{2.12}$$

where N_{docs} is the total number of documents and N_i is the number of documents containing term i. The overall weight of a term is the product of the local and global weight:

$$tf(i,j) * log_2(\frac{N_{docs}}{N_i}) \tag{2.13}$$

Log-Inverse Document Frequency (Log-IDF)

In the Log-IDF model, the local weight of a term is a logarithmic function of the term frequency given by:

$$log_2(tf(i,j)+1) \tag{2.14}$$

and its global weight is the Inverse document frequency given by equation 2.12. The overall term weight is:

$$log_2(tf(i,j)+1) * log_2(\frac{N_{docs}}{N_i}) \tag{2.15}$$

Log-Entropy (LET)

In the Log Entropy model, the local weight of a term is a logarithimic weight function given by equation 2.14 and the global weight is entropy-based given by:

$$G(i) = 1 - \frac{H(d|i)}{H(d)} \tag{2.16}$$

where

$$H(d|i) = -\sum_{k=1}^{J} p(i,k) * log_2 p(i,k) \tag{2.17}$$

is the entropy of the conditional distribution given term i, where there are $i = 1...I$ terms and $j = 1...J$ documents. H(d) = $\log_2$(J) is the entropy of the document distribution.

Term Frequency-Entropy(TF-Entropy)

In the TF-Entropy model, the local weight of a term is the term frequency, tf(i,j) and the global weight is the given by equation 2.16. The overall weight of a term is given by:

$$tf(i,j) * G(i) \tag{2.18}$$

2.3.2 Topic based Models

Bug reports may comprise of related topics or concepts that can give the reader an idea of the nature of the problem. If these topics cannot be explicitly defined, they can be identified by looking at the co-occurrence relationships between words across documents. Topic based techniques overcome the limitation of word based models which is the assumption that terms are independent.

Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI) assumes a latent structure in the usage of words for every document and recognizes topics[11] . LSI overcomes two shortcomings of traditional Vector Space Model approaches, synonymy and polysemy, by discovering relationships between terms across multiple documents. Given a term-document matrix, LSI outputs a reduction through a Singular Value Decomposition (SVD). SVD reduces the vector space model in less dimensions while preserving information about the relationship between terms. The dimension of the matrix after SVD is equal to the number of topics considered, k. Determining the optimal value of k for a problem is still an open research question. As such, if k is small, the topics are small and more general, whereas if k is large, the topics tend to overlap semantically.

Latent Dirichlet Allocation(LDA)

Following Probalistic Latent Semantic Indexing (PLSI) [17], a fully generative Bayesian model called Latent Dirichlet Allocation was introduced [6]. The underlying idea behind LDA is that documents can be represented as random mixtures over latent topics, each topic being characterized by a distribution over words. As with LSI, determining the the optimal number of topics is a challenge [6]. We experiment with a range of topics for both approaches in our study.

Random Projections/Random Indexing (RP)

Random Projections is an incremental vector space model introduced in [20], and is computationally less demanding as it does not require a separate dimensionality reduction step, unlike LSI. Each word is assigned a unique, random vector called an index vector which is sparse and has a high dimensionality, d. Each time a word occurs in a context(document), the context's d-dimensional index vector is added to the context vector for the word. In this way, each word is represented by a d-dimensional context vector, which is the sum of word contexts [40].

2.4 IR in Software Engineering

In recent years, IR techniques have been brought into the field of Software Engineering to analyze artifacts generated during the software development lifecycle. During software development, various related artifacts are created. These artifacts can be divided into 3 broad categories- structured data (e.g. analysis data), semi-structured information such as source code and unstructured information such as text in documentation, emails, code comments etc. A Software Engineering problem related to textual artifacts can be translated into an instance of a standard IR problem in a reasonably easy manner [12].

Dekhtyar et al. [12] claim that Software Engineering is unique in terms of the way IR methods are used in it. The size of the document collection for Software Engineering problems is significantly smaller than the size of a document collection in a standard IR application. As a result, this opens up the possibility to use a wide range of IR techniques such as Latent Semantic Indexing that scale poorly on large document collections but may provide decent results on Software Engineering problems.

Documentation such as requirements, design documents, test plans, user manuals, comments in source code, source code identifiers etc. consitute user-centric information. This information is semantic and is essential for developers to understand a software system. This section discusses the state of the art in the application of IR techniques in software development, particularly in software evolution and maintenance.

2.4.1 Requirements Engineering

As software product requirements are expressed in natural language, there are two problems that have been exploited by IR techniques i) searching for missing requirements and ii) linking requirements for release planning and prioritization.

Dag et al. [30] use what they refer to as Linguistic Engineering using Vector Space models to link product requirements to customer wishes. Their underlying assumption is that if referring to the same functionality, both sets of requirements use common terminology. They developed an Open source tool called Reqsimile [28] to facilitate requirements linkage. The primary goal of their work is to get a set of potential linkages between product requirements and customer wishes for new incoming requirements, which is generally accomplished in industry using search facilities that are cumbersome and prone to missing links.

2.4.2 Traceability Link Recovery between Software Artifacts

Traceability links help in estabilishing the relationship between software artifacts such as source code, documentation, test cases, bugs etc. As maintaining these linkages is cumbersome, IR based techniques such as Vector Space Models, probabilistic IR approaches and LSI have been used. The primary goal of traceability recovery for a given problem is to report potential linkages to the user with an optimal precision and Recall.

Antoniol et al. used a Probablistic IR model and a Vector Space IR model to construct links between source code and documentation [2]. Lormans et al. investigated the use of LSI to establish traceability between requirements, design and test cases [22]. Marcus et al. [24] used Vector Space model and LSI to recover traceability links between emails and source code.

2.4.3 Impact Analysis

The goal of impact analysis is to identify products affected by a proposed change [3]. For instance, adding new functionality to a system involves adding new functions, which starts at the requirements level and propagates through different artifacts to the source code. Antoniol et al. [1] use probabilistic IR techniques and Vector Space models to trace the text of a maintenance request to the set of affected system components. They apply these onto a public domain C++ library called LEDA (Library of Efficient Data and Algorithms) [15].

2.4.4 Concept Location

The goal of Concept Location is to identify parts of a software system that implement a specific functionality or concept. It is one of the most common activities in software maintenance and evolution and in program comprehension. Developers use search facilities like grep to locate concepts of interest in the source code or documentation. However, this has its limitations as it performs poorly when concepts are hidden implicitly in the code

[25]. Static and dynamic analysis tools have also been developed for concept location [35, 36]. Marcus et al. [25] use LSI to map concepts in natural language to source code and evaluate their results on a case study on the NCSA Mosaic [32].

2.4.5 Software Reuse

Software Reuse involves the use of existing software to build new software. Software reuse has the potential to improve quality, productivity, reliability and maintainability[47]. Ye et al. [54] develop a system called CodeBroker which provides personalized information to developers based on the task they are performing. Their system uses LSI. Hipikat [52] is another tool developed to assist developers by recommending relevant software artifacts (source code, documentation, bug reports, version information etc.) based on the developers' context. Hipikat also uses LSI to develop the underlying linkages.

2.5 Summary

This chapter presented various background concepts related to the subject of the thesis. We first described the structure and life-cycle of a bug report in Section 2.1. In Section 2.2, we outlined the major processing steps in Natural Language Processing. In Section 2.3, we discussed two main categories of IR models used in this work: word based and topic based models. For word based models, we discussed common global and local weighting schemes. Lastly, in Section 2.4, we looked at some applications of Information Retrieval in the field of Software Engineering.

Chapter 3

On Bug Reports: State-of-the-art Research

In this chapter, we survey the state-of-art research in Duplicate bug report detection. There have been several approaches proposed to help triagers in duplicate bug report detection. These can be broadly categorized into two approaches. The first approach is to prevent duplicate reports from reaching developers by automatically filtering them. The second approach deals with offering the end user a list of top-N similar bug reports, allowing the triager to compare the incoming bug report with the ones provided in the list. If the triager finds a report in the list that matches the incoming bug report, it is marked as a DUPLICATE of the existing report. In such an event, the existing bug report is considered as the master report.

3.1 Providing a List of Top-N Similar Bug Reports

In this approach, the end user is provided a list of top-N similary bug reports to assist with the task of duplicate bug report detection. The end user could be a triager or a user who is interested in determining whether a duplicate exists for his query. The quality of the list is crucial.

3.1.1 Using Natural Language Processing

Runeson et al. [39] apply the Vector Space model in order to detect duplicate reports, where each bug report is represented as a vector, and each term in the vector is given a weight:

$$weight(term) = 1 + log_2(tf(term)) \tag{3.1}$$

They experiment with bug reports belonging to Sony Ericcson Mobile Communications. They found that 90% of duplicate bug reports are submitted in the range between 20 days forward and 60 days back, they call this a **time frame**. They create a domain specific thesaurus to substitute tokens with their synonyms and perform stemming, stopword removal and spell checking on their data. They evaluate their results on different top list sizes of 5, 10 and 15 and record the Recall rate with the following parameters:

- with and without using the project field of the bug report
- with and without using synonyms that they generate from a domain specific thesaurus
- using stop word lists of different sizes
- different similarity measures- Jaccard, Cosine and Dice
- weighting the summary of the bug report more than the regular description

Their results indicate that not all duplicate bug reports can be detected using their approach alone. At best, a maximum of 60% duplicates can be found, with a very large top list size. They obtain best results of finding 39% reports for a top list size of 10 and 42% for a top list size of 15 across different variants of similarity measures, stop lists, spell checking and weighting of the summary.

3.1.2 Using Natural Language and Execution Information

Wang et al.[53] investigate the use of execution information of bug runs in addition to Vector Space models, where each term in the vector has a weight given by:

$$weight(word) = tf(term) * idf(term) \tag{3.2}$$

As there is no execution information in bug reports in Eclipse and Mozilla, they manually create execution traces by reproducing the bugs using details in the bug reports. They evaluate their approach on 1492 bug reports from Firefox. Their results show that with top lists of 1-10, they achieve a Recall rate of 67-93% as opposed to 43-72% using NLP techniques only.

The drawback to their approach is the cost of using execution infomation as it implies additional storage in bug repositories to store execution information. They also point out that there is a burden posed on bug reporters to run an instrumented version of the software and submit execution information.

3.1.3 Using Word Semantics

Rus et al. [51] used the WordNet [50] lexical database which groups words with similar meanings into synonymous sets. Each set in turn defines a concept. A word may belong to more than one set, if it has more senses. In WordNet, similarity between concepts is measured in terms of the distance (path length) between concepts. The larger the distance, the less similar the concepts. They created an experimental data set from Mozilla's Hot Bugs List and for 20 bugs they retrieved 50 duplicates each, thus generating 1000 paris of main and duplicate bug reports, and another set of 1000 non-duplicate pairs of defect reports.

In Wordnet, word sense disambiguation is done by two techniques:

- Sense I which maps each word to its most common sense
- All-Senses which maps each word to all its senses

They experimented with 10 different WordNet similarity measures. They found that the accuracy for both Sense I and All-Sense techniques were very similar, indicating that word sense disambiguation is not important for identifying duplicates. Using their approach, they achieved a Recall of 64.08% with the LCH measure that primarily deals with nouns, supporting the notion that using mostly nouns for bug duplicate detection is enough for achieving a good Recall rate.

3.1.4 Using a Discriminative Model Approach

In [45], Sun et al. use discriminative models to detect duplicate bug reports. They consider bug report duplication as a binary classification problem and classify bug reports into duplicates and non-duplicates. They applied their approach to 3 large open source projects

- Open Office (with 12732 bugs over a year), Eclipse (with 44,652 bugs over a year) and on Firefox (with 47,304 over 2002 to 2007).

They apply the SVM classifier to determine how likely two reports are to be duplicates of each other and they retrieve a list of candidates based on the probabilities. They organize the bug reports into "buckets", a hash-map like data structure, in which the master report is the key and all of its duplicates are values. If an incoming report is a duplicate, it is added to its corresponding "bucket", and if its not a duplicate, a new bucket is created for which the new bug report becomes the main report.

They extract 2 sets of examples from the buckets- positive examples corresponding to pairs that are duplicates and negative examples corresponding to non-duplicate pairs. They also extract 3 different bag of words from each report- summary, summary and description, and description only, in such a manner that makes many combinations of "features" possible. They use an idf weighting scheme over the summary corpora, description and both summary and description. Altogether, they extract a total of 27 features from each pair of bug reports. They also consider bigrams as features, which increases the number of features to 54. All these features are then input to the SVM to build a discriminative model.

When a new bug report arrives, an iteration over all the buckets is done and the similarity between the bug report and each bucket is calculated. In the end, a list of master reports with the highest similarity is returned. Their techniques outperformed the state of the art results by 17-31% on Open Office, 22-26% on Firefox and 35-43% on Eclipse.

3.2 Automatically Filtering Bug Reports

In this approach, the key premise is to filter bug reports automatically and in this manner prevent duplicates from reaching the developers. Therefore, this approach requires little or no triager involvement and one would have to assume that the filtering performed is accurate.

3.2.1 Using Textual Semantics and Graph Clustering

In [18], Jalbert et al. use a combination of textual similarity, surface features (such as bug severity, operating system information etc) and graph clustering algorithms to identify bug duplicates. Their data set consists of 29000 bug reports from Mozilla spanning over 8 months. For every bug report that arrives, their proposed model uses features from the bug report to predict whether it is a duplicate or not. They employ a linear regression model to make a binary classifier to distinguish between duplicate and non-duplicate bug reports.

On their data set, they found that IDF is not effective in distinguishing duplicate bug reports from non-duplicates. Therefore, they use a different term-weighting scheme for the vector representation:

$$weight(word) = 3 + 2 * log_2(tf(term)) \tag{3.3}$$

They use a cosine similarity measure to get the top-N similar bug reports. Using their proposed textual similarity metric, a graph is derived upon which they apply a graph clustering algorithm to get a set of clustered reports. The underlying premise of their approach is that if an incoming bug report does not belong to any cluster, it is less likely to be a duplicate, whereas a bug report with its many duplicates can be found in a cluster.

With their technique, they were able to detect 8% of the duplicates, and achieved a Recall rate improvement of 1% over Ruenson's best Recall rate. An important finding they made was that semantically rich textual information contributed more than surface features in detecting duplicates. The title (summary) and the bug description were deemed the most important distinguishing features.

3.3 Summary

In this chapter we review the state-of-the art research in the area of duplicate bug report detection. The two types of approaches used in the literature have been discussed, namely providing the end user a list of top-N bug reports and automatically filtering bug reports. Both approaches have their pros and cons. However, most of the works have focused on ways of improving the top-N list of bug reports. Evidently, all the works discussed have extensively relied on the usage of Information Retrieval techniques, using Natural language Processing only or a combination of Natural Language Processing in conjunction with other approaches.

Chapter 4

Proposed Approach

In this chapter, we discuss our proposed approach by defining the heuristics used for the case studies and describing the process to determine bug report similarity. We outline the evaluation measures used. Later on, we discuss the importance of using an optimal time frame for duplicate bug report detection and present some findings from historical bug report data from the NetBeans and Eclipse projects. Lastly, we introduce our four research questions that we address in the following chapter.

4.1 Heuristics

We retrieve textual information from the target duplicate bug reports using the heuristics defined below. The purpose of using these heuristics is to assess which types of data contribute most to determining bug report similarity.

- Double Weighted summary and long description: Runeson et al. [39] suggest that the bug report summary should be treated twice as important as the long description. Therefore, we extract the free-form text in the bug report summary and give it double weight. In other words, for a term appearing in the summary once, it is counted as occurring twice and a term appearing in the long description once is counted only once.

- Summary only: In order to see the effectiveness of using terms in the summary field, only the summary from each bug report is extracted.

- Long description only: In order to see the effectiveness of using terms in the long description, only the long description from each bug report is extracted.
- Equally weighted summary and long description: In this case, both the summary and long description with equal weights are extracted.

Wherever appropriate, we have combined the above three heuristics with the ones described below:

- Use of partial stack trace: In their case study on the Eclipse project, Schroter et al. found that up to 60% FIXED bug reports that contained stack traces involved changes to one of the stack frames [42]. Moreover, a defect was typically found in one of the top-10 stack frames of a bug report. We consider the stack frames contained within the long descriptions of the bug reports. We extract identifiers from the top 10 stack frames if there is any stack trace embedded within the bug report's description. We do not consider stack traces in attachments since some bug tracking systems such as the one in NetBeans detects patterns in the stack traces contained in attachments. We conjecture that identifiers from stack frames can help discriminate duplicate bug reports from non-duplicates.
- Surface features: Features such as the Component, Product and Classification of the bug report are also extracted.

4.2 Determining Bug Report Similarity

We begin by defining a time frame within which we want to extract bugs. Bug reports from the chosen time frame are extracted from the Bugzilla repository in XML format, where each bug report contains elements for different attributes in the report such as the Summary, Product, Component etc. Figure 4.1 illustrates a block diagram of the steps to determine bug report similarity and calculating the Recall Rate.

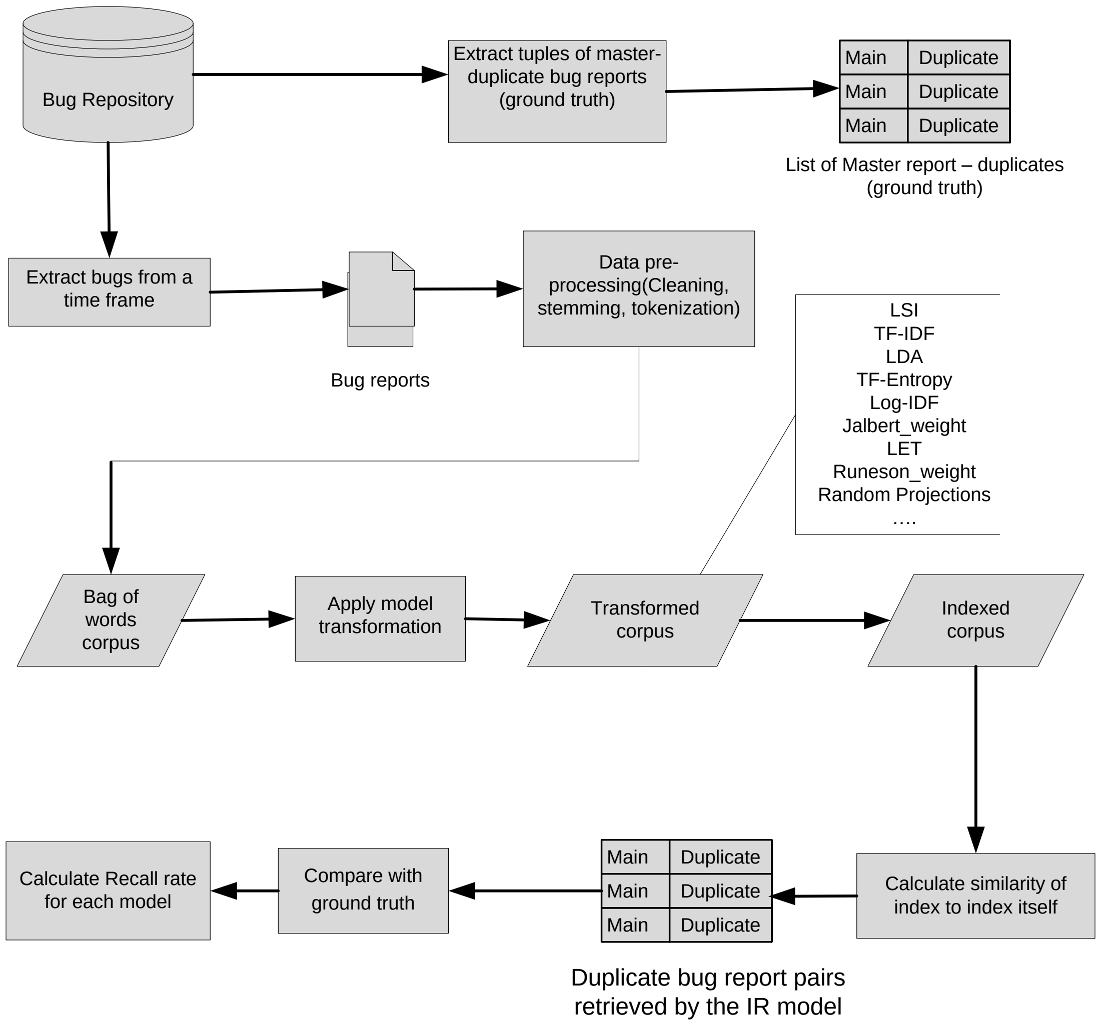
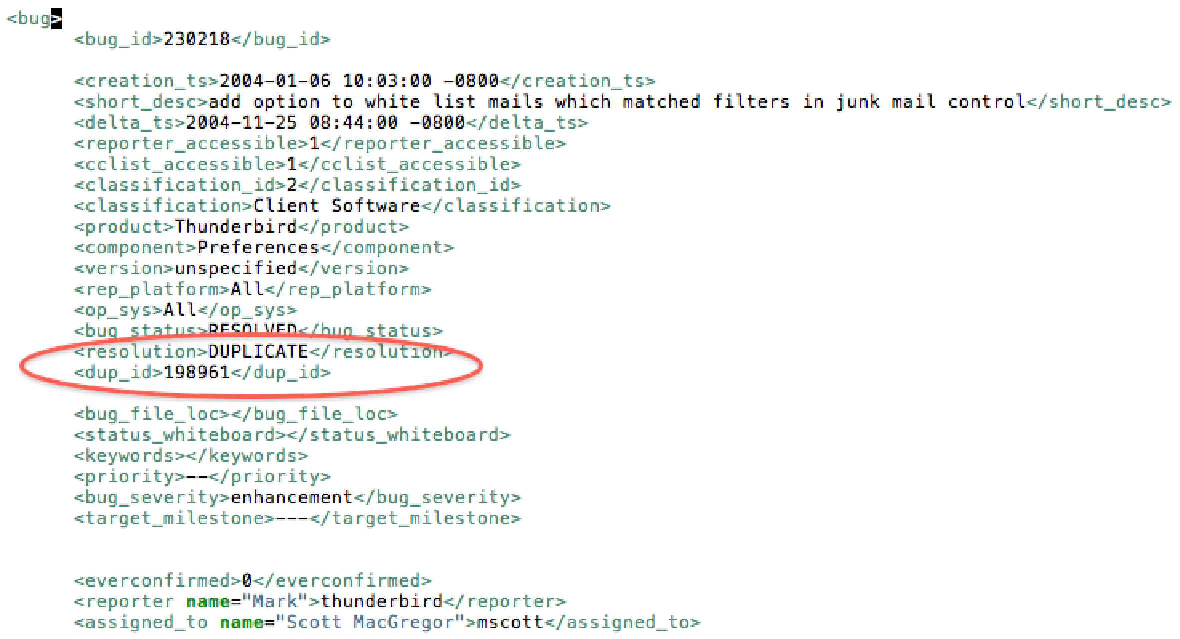

Bug Repository
Extract tuples of master-duplicate bug reports (ground truth)
Main | Duplicate
Main | Duplicate
Main | Duplicate
List of Master report – duplicates (ground truth)
Extract bugs from a time frame
Bug reports
Data pre-processing(Cleaning, stemming, tokenization)
LSI
TF-IDF
LDA
TF-Entropy
Log-IDF
Jalbert_weight
LET
Runeson_weight
Random Projections
….
Bag of words corpus
Apply model transformation
Transformed corpus
Indexed corpus
Calculate similarity of index to index itself
Main | Duplicate
Main | Duplicate
Main | Duplicate
Duplicate bug report pairs retrieved by the IR model
Compare with ground truth
Calculate Recall rate for each model

Figure 4.1: Block Diagram of Process to Determine Bug Report Similarity



```
<bug>
    <bug_id>230218</bug_id>

    <creation_ts>2004-01-06 10:03:00 -0800</creation_ts>
    <short_desc>add option to white list mails which matched filters in junk mail control</short_desc>
    <delta_ts>2004-11-25 08:44:00 -0800</delta_ts>
    <reporter_accessible>1</reporter_accessible>
    <cclist_accessible>1</cclist_accessible>
    <classification_id>2</classification_id>
    <classification>Client Software</classification>
    <product>Thunderbird</product>
    <component>Preferences</component>
    <version>unspecified</version>
    <rep_platform>All</rep_platform>
    <op_sys>All</op_sys>
    <bug_status>RESOLVED</bug_status>
    <resolution>DUPLICATE</resolution>
    <dup_id>198961</dup_id>

    <bug_file_loc></bug_file_loc>
    <status_whiteboard></status_whiteboard>
    <keywords></keywords>
    <priority>--</priority>
    <bug_severity>enhancement</bug_severity>
    <target_milestone>---</target_milestone>


    <everconfirmed>0</everconfirmed>
    <reporter name="Mark">thunderbird</reporter>
    <assigned_to name="Scott MacGregor">mscott</assigned_to>
```

Figure 4.2: Bug Report in XML format

- For any bugs with the "Resolution" field set to "DUPLICATE", there exists an additional field called "dup id" which is the master bug report. Figure 4.2 shows a bug report its raw XML format. Here, Bug 230218 has been marked as a duplicate of Bug 198961, which is the master bug report. For each bug report, we check whether its resolution was "Duplicate" and we retrieve the dup id and search for this dupid in our data set. If the master report is found in our data set, implying that both the duplicate and its master bug report were created in the same time frame, we save this duplicate and master report tuple in a list for future reference. This set of all master-duplicate bug report tuples would form our ground truth for the Recall rate calculation.

- From each bug report in the data set, we extract the free-form text depending on the heuristics being used, and perform some data-preprocessing such as cleaning, stemming, stopword removal and tokenization, as described in Section 2.2. For all the text extracted from the bug reports, we create a bag of words corpus on which we apply a model transformation. The model transformation transforms the bag of words corpus into the space of the desired model. From this transformed corpus, we generate an index of documents and terms.

We do not distinguish queries from the main corpus, in other words, we are simply interested in determining which IR model and heuristics yield the best Recall rate. We retrieve the top-N links or recommendations in order of their cosine similarity value. Finally, the Recall rate is calculated for each IR model for a set of heuristics and parameters (number of links retrieved, num of topics, if relevant) by comparing the links retrieved with the ground-truth. In the end, we compare the performance of the models in terms of Recall rate and determine which model performs best for a given set of heuristics and chosen top list size.

4.2.1 Size of Top list

Previous works [39, 53] have shown that using a top list size of 1 is too conservative as the Recall rate achieved is poor. Instead, a top list size of 7±2 could be reasonable, as Miller [26] shows in his work on the human capacity for processing information. However, the suitability of this number would have to be experimentally verified when such a system is put into production.

4.3 Evaluation Measures

For measuring the similarity between two documents, the cosine similarity measure given in equation 4.1 has been used. In [39], Runeson et al. found that the Dice and Jacard coefficient did not improve the Recall rate over the cosine similarity measure. Therefore, for evaluation we only use the cosine similarity measure.

$$cos\theta = \frac{A \cdot B}{||A||||B||} \tag{4.1}$$

Our end goal is to find how many duplicates were detected by our framework. Although such type of retrieval is traditionally evaluated by metrics such as Precision and Recall, these metrics do not particularly fit well in the problem of duplicate bug detection.

For example, if we were to search for a duplicate in a top list size of 10, we are only really interested in knowing whether we can find the Master bug report in the list of size 10. For this, if we retrieved the Master bug report, we would achieve a Precision of 10%

and a Recall of 100%. Now lets say we could not retrieve the Master bug report in the top 10 list, our Recall would be 0%. In other words, our Recall would always be either 0 or 100%. This does not entirely indicate how well we performed.

Natt och Dag et al. propose Recall Rate [29], which is later used as a metric by Runeson et al.[39] and other works.

$$RecallRate = \frac{N_{recalled}}{N_{total}} \tag{4.2}$$

where N$_{recalled}$ refers to the number of duplicate bug reports whose master reports are retrieved for a given top-list size and N$_{total}$ is the total number of duplicate bug reports. Recall Rate has been defined as the percentage of duplicates for which the duplicate was found. In other words, it is a measure of the total duplicates found using the approach employed.

4.4 Time Frame of Duplicates

Due to the large volume of incoming bug reports in projects like Eclipse and Firefox, it is obvious that for an incoming bug report the set of candidate bug reports to search against is large. Runeson et al. propose an approach for narrowing the search space for duplicate bug detection by using defect reports created within a certain time frame [39]. They introduce the idea of "time frame" which can be specified by a user. In their study on bug reports from Sony Ericcson, they find that while 53% of duplicates were submitted within 20 days after their master report, 90% are submitted in between 20 days forward and 60 days back.

Based on this observation, Wang et al. [53] used this window in their study. In their data set of reports submitted in 3 consecutive months, they treated bug reports filed in the first 50 days as existing reports and the remainder as new bug reports [53]. However, Runeson's data was from an industrial case study and the applicability of a time frame of 50 days or so in the open source projects that follow a different paradigm of software development has not been investigated so far. To determine the time frame for open source projects for a certain level of desired coverage, we gathered some statistics about related to duplicate bug reports in 2 major open source software projects - Eclipse and NetBeans.

4.4.1 Experimental Setup

For both Eclipse and NetBeans, we obtained the sql dumps of the Bugzilla repositories from a previous MSR Challenge 2011[38]. For Eclipse, we consider data from a period of October 2001 to June 2010, while for NetBeans the data spans from June 1998 to June 2010. We collected information of tuples of master reports and their corresponding duplicates such as the creation date, status, priority, product, component, operating system etc. We run into the reverse duplicate problem in which the date of creation of the master bug report is ahead of the date of creation of its duplicate. In such cases, we still treat the master report as the master since is linked to several duplicates.

Based on the date of creation of master bug reports and their corresponding duplicates, we compute number of days before and after the submission of the main report within which a certain percentage of duplicates can be found. We also gather statistics about the percentage of duplicates in which the master report and the duplicate have different features such as the status, priority, component, product, operating system.

4.4.2 Discussion on Obtained Results

Some characteristics of the data we collected for NetBeans are outlined in Table 4.1. The box plot of the distributions of duplicates from the date of submission of the main report(corresponding to Day 0) and the cumulative frequency distibution of duplicates over time for NetBeans over a period of 12 years are shown in Figure 4.3 and 4.4 respectively.

Collected Characteristic	**Value**
Total number of bugs	185578
Total number of duplicates	27086
% of duplicates belonging to different products	19%
% of duplicates belonging to same product but different components	19%
% of duplicates with different priority	46%
% of duplicates with different severity	2%
% of duplicates with different different rep platform	36%

Table 4.1: Characteristics of Duplicate Bug Reports in NetBeans

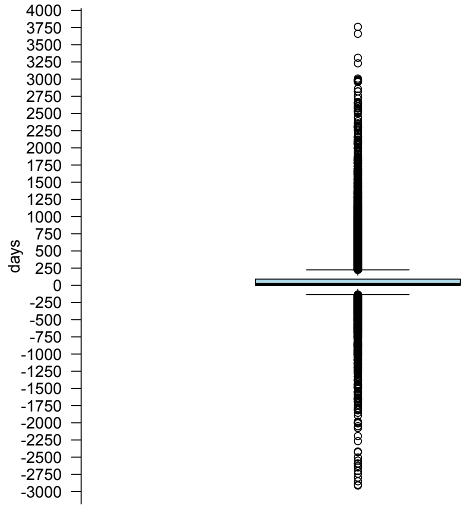


Figure 4.3: Boxplot of Distribution of Duplicates in NetBeans

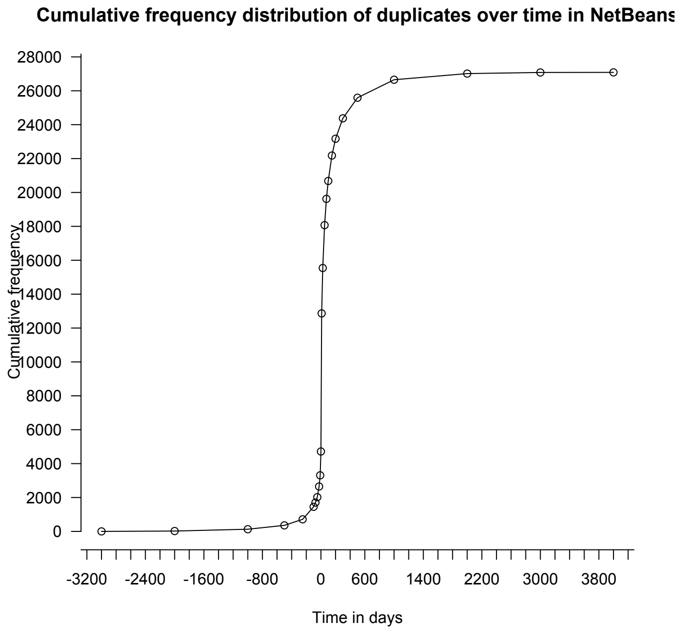


Figure 4.4: Cumulative Frequency Distribution of Duplicates Over Time in NetBeans

The charactersitics of the data set for Eclipse are summarized in Table 4.2.

Collected Characteristic	**Value**
Total number of bugs	316911
Total number of duplicates	37753
% of duplicates belonging to different products	11%
% of duplicates belonging to same product but different components	14%
% of duplicates with different priority	23%
% of duplicates with different severity	41%
% of duplicates with different different rep platform	21%

Table 4.2: Characteristics of Duplicate Bug Reports in Eclipse

The box plot of the distribution of duplicates for Eclipse and the cumulative frequency distribution of duplicates over time for Eclipse over a period of 9 years are shown in Figure 4.5 and 4.6 respectively.

We observe similar trends in the cumulative frequency distribution of the two projects with a notable sharp increase in the cumulative frequency in the period around Day 0 which flattens after some amount of time.

The percentiles of the duplicate bug reports that can be found before/after the creation of the master report (Day 0) for NetBeans and Eclipse are given in Table 4.3 and 4.4.

Percent	0%	10%	15%	20%	30%	40%	50%	60%	70%	80%	85%	90%	100%
Days	-2913	-24	-3	0	0	4	13	30	62	131	190	300	3760

Table 4.3: Percentiles for NetBeans

Percent	0%	5%	10%	20%	30%	40%	50%	60%	70%	75%	80%	90%	100%
Days	-2913	-219	-55	0	0	3	14	38	90	135	201	463	3176

Table 4.4: Percentiles for Eclipse

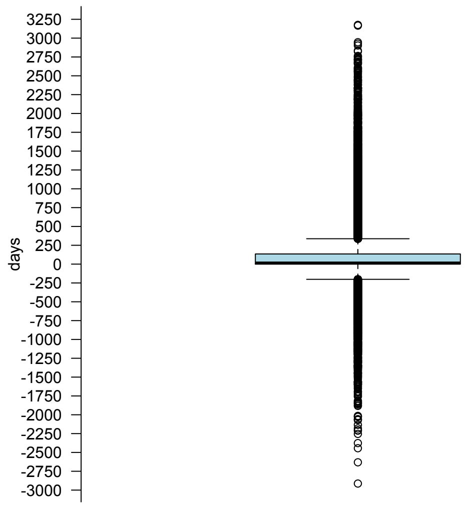


Figure 4.5: Boxplot of Distribution of Duplicates in Eclipse

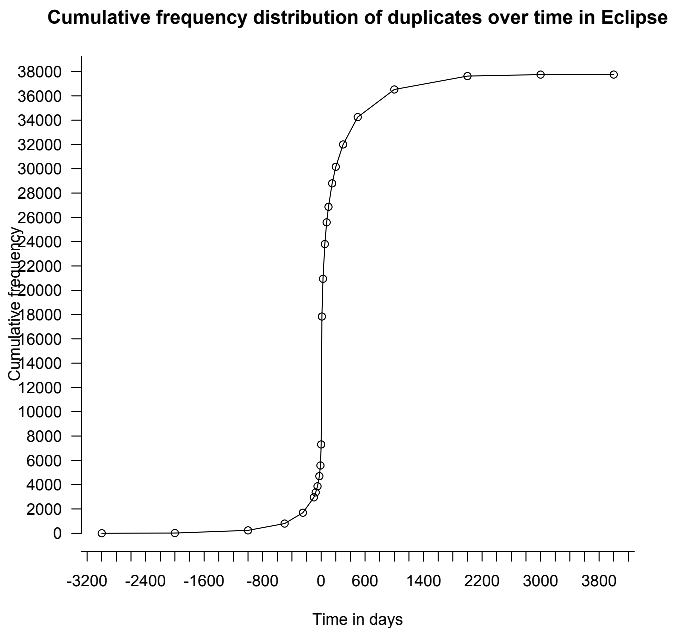


Figure 4.6: Cumulative Frequency Distribution of Duplicates Over Time in Eclipse

It is evident from the data that in NetBeans, we can cover upto 75% of the duplicates between a period of 3 days before and 190 days (a little above 6 months) after the submission of the main bug report. While for Eclipse, we can cover 70% of the duplicates between a period of 55 days before and 201 days after the date of submission of the master bug report. Based on this, for our case studies, we use bug reports spanning 6 months.

Aside from determining a suitable time frame to maximise the number of duplicates and master reports that can be found together, we also looked into the possibility of incorporating other attributes from the bug reports such as the operational environment information and bug priority, severity etc.

- **Bug Priority**: In the bug repositories of both projects, bugs are assigned a priority from P1 to P4, the former implying highest priority. As outlined in Table 4.2 and 4.1, the percentage of duplicate pairs with different priority in Eclipse is 23% and 46% in NetBeans. A good explanation for this difference is that assessing the bug priority is highly subjective.

- **Product and Component**: 19% master reports are assigned to a different product than their corresponding duplicate bug report in NetBeans, while in Eclipse 11% master - duplicates are assigned to different products. Furthermore, for those master-duplicate bug report pairs with the same product, 19% had different components in NetBeans and 14% in Eclipse.

- **Bug Severity**: Bug severity is initially assigned by the bug reporter who gives the bug a severity of one of the following: Minor, Major, Enhancement, Trivial, Blocker, Normal and Critical. Interestingly, only 2% of master - duplicate bug report pairs in NetBeans have different severity. Upon investigating the data closely, an interesting finding was made: 92 % of the duplicates in NetBeans have a severity of "BLOCKER". This seemed unusually high. Furthermore, 89% of all bugs in NetBeans have a priority of "BLOCKER". On the other hand, in Eclipse we got a more reasonable distribution, with 41% of duplicate pairs having different bug severity. Moreover, 64% of all the bugs in Eclipse have a severity of "NORMAL" and only 2% "BLOCKER". Clearly, this shows a big difference in the way the two projects use the bug severity field. Based on these findings, we can say that these two attributes cannot contribute substantially in improving our duplicate detection as we did not find a useful trend in the usage of these fields in the context of bug duplication.

4.5 Research Challenges

The previous techniques proposed to help triagers with the process of duplicate bug detection as discussed in chapter 3. These works have relied on the usage of IR techniques, using NLP only or a combination of NLP and other approaches. To the best of our knowledge, there has been little work done to do an exhaustive comparison of the performance of different IR models (especially using more recent advances in IR such as topic modeling) on this problem and understanding the effectiveness of different heuristics across various application domains. To this end, in this thesis, we propose and investigate four research questions:

RQ1: *What is the impact of applying different heuristics in helping the detection of duplicate bug reports across different application domains?*

RQ2: *Based on the selected heuristics, how does the performance of topic based IR models compare with that of word based models?*

RQ3: *Among word based and topic based models, which ones are the most appropriate for use in the context of duplicate bug report detection?*

RQ4: *For a required coverage of duplicate detection, what is the optimal time frame for its query space?*

4.6 Summary

In this chapter, we have detailed our proposed approach for detecting duplicate bug reports. Section 4.1 outlines the heuristics we incorporate for our study. Section 4.2 presents an overview of the process to determine bug report similarity and evaluate the IR models. Section 4.3 defines the evaluation measures used. Section 4.4 discusses the signifance of time frame in the context of duplicate bug report detection and presents some findings from historical data from NetBeans and Eclipse. Finally, Section 4.5 outlines four research questions that we investigate through our experimental studies.

Chapter 5

Experimental Studies

In this chapter, we present the results of our experimental studies. Section 5.1 discusses the data sets and the experimental setup for the two case studies. This is followed by a discussion on the obtained results. Section 5.2 presents an online framework for duplicate bug report detection which draws upon findings of the case studies and incorporates the idea of a shifting time frame for the search space of incoming bug reports. Lastly, Section 5.3 outlines the threats to validity to our experimental studies.

5.1 Case Studies

We perform our experiment on bugs from repositories of two major open source software projects- Eclipse and Firefox. While Eclipse is a popular IDE written in Java, Firefox is a web browser written in C/C++. Both these projects are very active and have large bug repositories.

5.1.1 Data Set

From the chosen bug repositories, we select a subset of bug reports for our experiments. We use all the 4330 bug reports from Eclipse's Platform project from the period of January 2009 to October 2009, and from Firefox we extract all 9474 bug reports from the period of January 2004 to April 2004. Table 5.1 provides details of the two data sets.

We have specifically selected bug reports from a period in the sufficient past since in choosing recent bug reports, it is likely that the resolution of the bug reports will change

Project	**Time Frame**	**Total number of bug reports**	**Number of duplicates within the Time Frame**
Eclipse	Jan 2009 - Oct 2009	4330	265
Firefox	Jan 2004 - April 2004	9474	667

Table 5.1: Summary of Datasets

and a lot of bug reports may still be pending resolution. This may affect our ground-truth, and in turn impact the accuracy of our results.

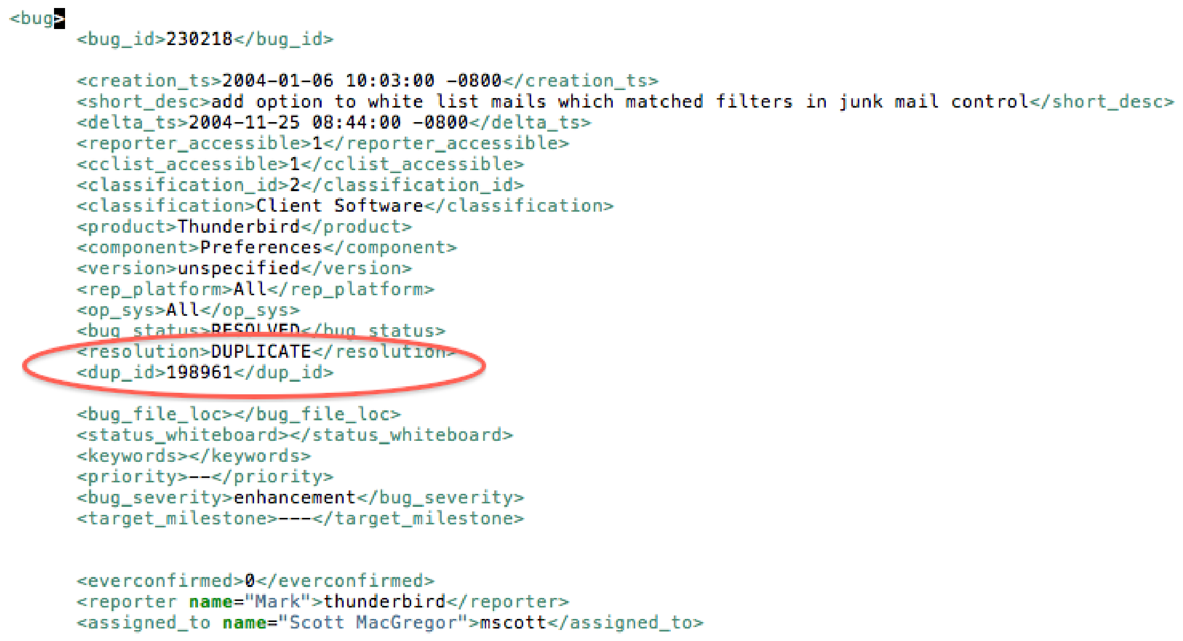
A bug report is marked INVALID when it cannot be reproduced. Before being marked INVALID, a bug report is triaged and may be assigned to a developer for investigation. Therefore, in our data sets, we do not discard invalid bug reports since a triager does not know of their nature when they are initially received. As such, for a practical duplicate bug recommender system, it is reasonable to treat all incoming bug reports alike, removing INVALID bug reports from the data set will introduce bias.

The bug reports are extracted in XML format. Each bug report has an element for the short description (summary), long description, creation date, product, component, resolution and so on. We parse the contents of XML elements, depending on the type of heuristics we want to incorporate. It is imperative that we extract and consider only the initial comments from the reporter of the bug in the long description, since that is the information available in the report when the bug is triaged initially. The comments in the long description that follow later are ignored for the purposes of the experiment.

The data set contains pre-annotated bug reports marked as duplicates. It is not uncommon to find some duplicate bug reports whose master report is not in the data set if the master report was created prior to the time frame we consider. To overcome this problem, we consider only those master-duplicate pairs whose date of creation lies within the time frame chosen for our study. In this way, we form our ground truth with these master-duplicate bug report pairs within the chosen time frame. There are a total of 265 master-duplicate bug report pairs for Eclipse and 667 such pairs for Firefox.

5.1.2 Experimental Setup

Each bug report is in XML format as shown in Figure 5.1. Using Python's xml.dom module, which is a light-weight implementation of the Document Object Model, we parse the bug reports in XML format to extract the elements such as the summary, long description,



```
<bug>
    <bug_id>230218</bug_id>

    <creation_ts>2004-01-06 10:03:00 -0800</creation_ts>
    <short_desc>add option to white list mails which matched filters in junk mail control</short_desc>
    <delta_ts>2004-11-25 08:44:00 -0800</delta_ts>
    <reporter_accessible>1</reporter_accessible>
    <cclist_accessible>1</cclist_accessible>
    <classification_id>2</classification_id>
    <classification>Client Software</classification>
    <product>Thunderbird</product>
    <component>Preferences</component>
    <version>unspecified</version>
    <rep_platform>All</rep_platform>
    <op_sys>All</op_sys>
    <bug_status>RESOLVED</bug_status>
    <resolution>DUPLICATE</resolution>
    <dup_id>198961</dup_id>

    <bug_file_loc></bug_file_loc>
    <status_whiteboard></status_whiteboard>
    <keywords></keywords>
    <priority>--</priority>
    <bug_severity>enhancement</bug_severity>
    <target_milestone>---</target_milestone>


    <everconfirmed>0</everconfirmed>
    <reporter name="Mark">thunderbird</reporter>
    <assigned_to name="Scott MacGregor">mscott</assigned_to>
```

Figure 5.1: Bug Report in XML Format

creation date, product, component etc. For the data cleaning and pre-processing steps, we use the Natural Language ToolKit (NLTK)[49], a group of open source Python modules available for research in natural language processing and text analytics.

- Stopword removal: We incorporate a list of stopwords, which consists of words from the SMART[21] stopword list, Java keywords and common Java identifiers. By doing this, we remove common words that have high frequency but little semantic content.
- Tokenization: We use NLTK's WhiteSpaceTokenizer module to tokenize text into tokens, or lists of substrings, in the process of which punctuation gets removed.
- Stemming: After stopword removal and tokenization, we stem each word using an implementation of the Porter stemmer algorithm in the NLKT.
- Treatment of Exception traces: As per [42], the top 10 exception stack frames of bug reports were deemed more important in bug resolution. Therefore, wherever possible, we extract the top 10 stack frames from the long description of the bug report.

In order to calculate the similarity between the bug reports using the techniques discussed in Section 4, we use Gensim [37], a Python framework that extracts semantic topics from documents. While Gensim supports TF-IDF, LET, LSI and LDA, we implement the rest of the techniques to the best of our knowledge.

5.1.3 Evaluation Measures

We use Recall rate, as defined by Natt och Dag et al. [31] and adapted by Runeson at al. in [39] as a measure of performance.

$$RecallRate = \frac{N_{recalled}}{N_{total}} \tag{5.1}$$

where $N_{recalled}$ refers to the number of duplicate bug reports whose master reports are retrieved for a given top-list size and N_{total} is the total number of duplicate bug reports. Recall rate measures the accuracy of the system in terms of the percentage of duplicates for which their master report is found in the top-N results. The Recall rate metric better suits the nature of our problem as it overcomes the limitation of the standard Recall measure that would be a binary value, either 0%(not found) or 100%(found) in this context [39].

5.1.4 Discussions on Results Obtained

In this section, we revisit the first three research questions and address them using the results obtained. Figure 5.2 and Figure 5.3 show the Recall rate obtained using individual local and global weights as well as a combination of global and local weights on the Eclipse data set. These plots correspond to the word based models. Figure 5.4 shows the Recall rates obtained using topic based models.

Figure 5.5 and Figure 5.6 show the results from the word based approaches using individual global and local weights, and a combination of global and local weights respectively on the Firefox data set. Figure 5.7 shows the Recall rate obtained using topic based approaches. In all the figures, the horizontal axis represents the list size and the vertical axis represents the Recall rate.

RQ1: *What is the impact of applying different heuristics in helping the detection of duplicate bug reports across different application domains?*
In the Eclipse dataset, using a combination of a double weighted summary, long description and exception stack trace information performs better than using just a double weighted

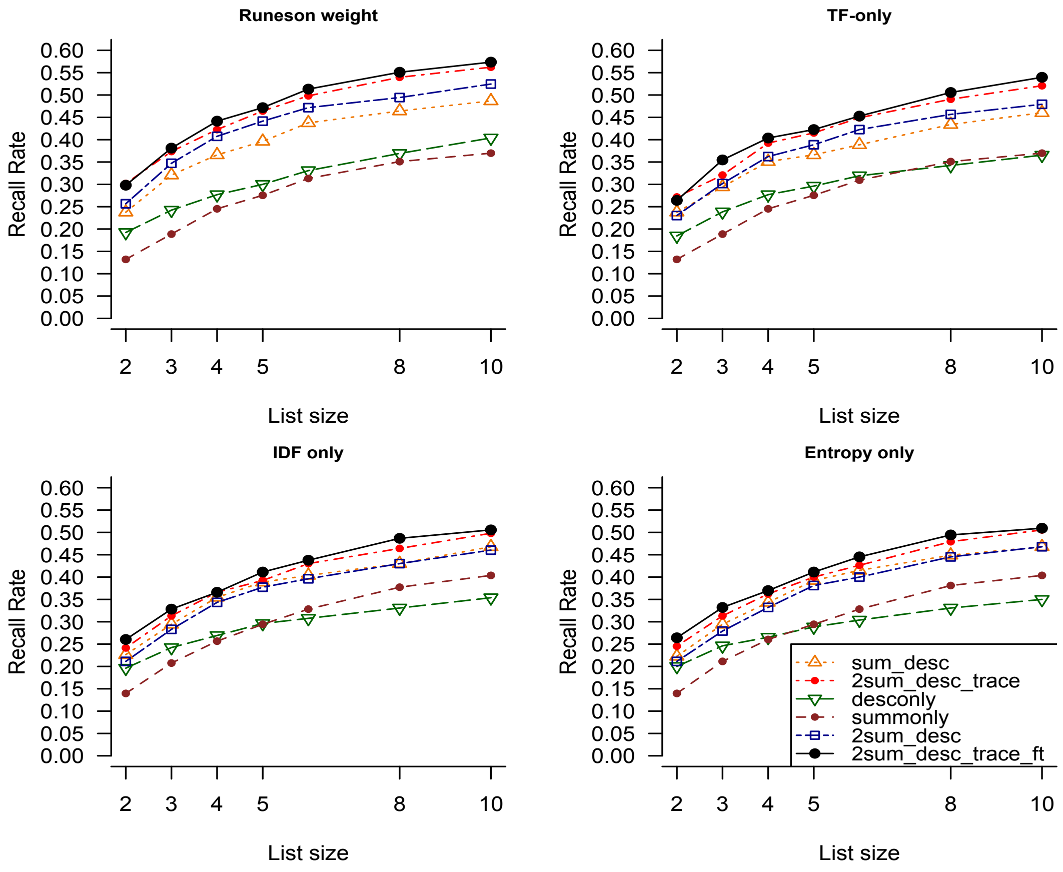


Figure 5.2: Recall Rates on Eclipse Using Local and Global Weights Individually (Word Based)

summary and description. However, using surface features does not increase the Recall rate. Futhermore, both these heuristics yield better results than using just the summary or the description. In word based models, using the description alone performs better than summary only. We speculate that this is because summaries tend to be short and carry less information to enable a model to discriminate duplicates from non-duplicates. However, in topic based approaches this does not hold as for both LSI and LDA, the summary only outperforms the description, but only for higher list sizes

In the Firefox dataset, we observe that using exception stack trace information has little or no effect over using a double summary and description only. This suggests something about the nature of the bug reports for this project. As compared to Eclipse, the

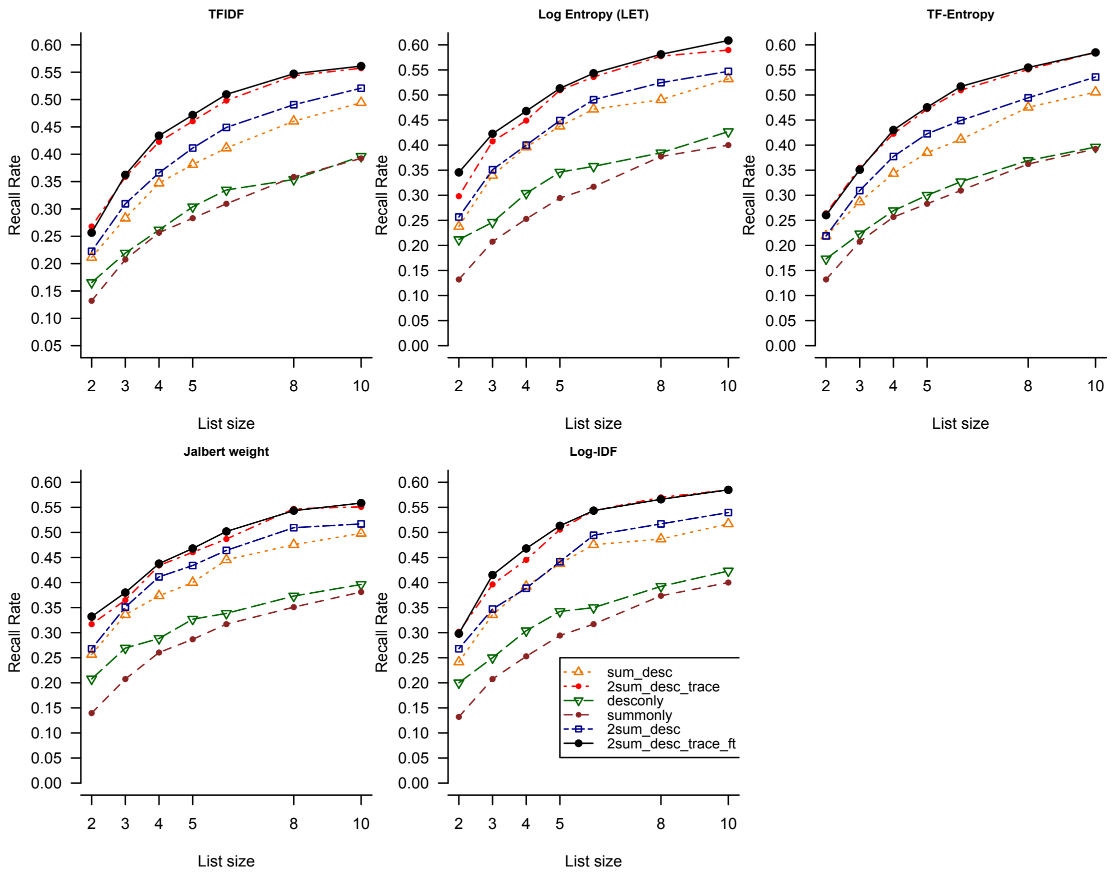


Figure 5.3: Recall Rates on Eclipse Using a Combination of Global and Local Weights(Word Based)

bug reports in the Firefox dataset tend to be more verbose and contain more descriptive information rather than stack frames. Therefore, using stack frames is not useful for this project. Also, the surface features help improve the Recall rate for some models, but not significantly.

RQ2: *Based on the selected heuristics, how does the performance of topic based IR models compare with word based models?*

For both LSI and LDA, we experimentally determine the optimal number of topics to be 550 and 400 respectively for the Eclipse dataset. For Firefox, we determined the optimal number of topics as 500 and 400 for LSI and LDA respectively. We use these topic counts

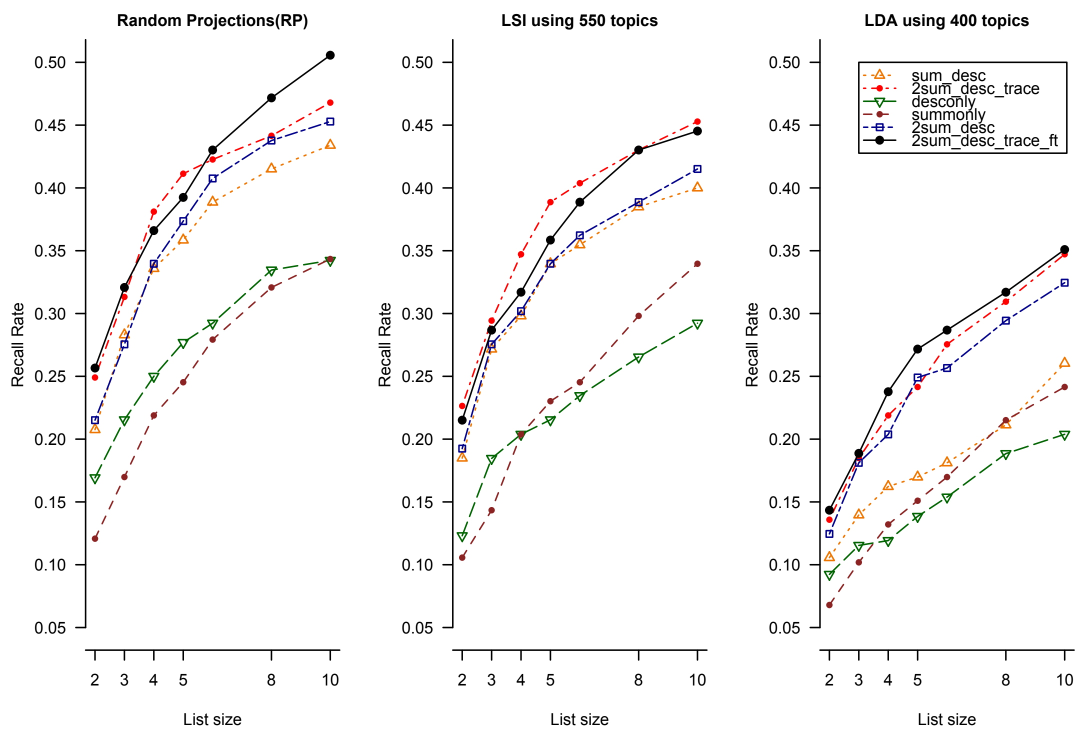


Figure 5.4: Recall Rates on Eclipse Using Topic Based Models

for the evaluation.

In both the case studies, word based models outperformed topic based ones. In Eclipse, as seen in Figure 5.4, for a list of size 2-10 the best Recall rate achieved with topic based models is between 0.25-0.5, whereas the best Recall rate achieved by word best techniques is between 0.35-0.6, an improvement of 10%.

Similarly, in Firefox, as illustrated in Figure 5.7, for a top size list of 2-10 the Recall rate for topic based approaches is 0.28-0.46 and for word based ones it is 0.31-0.58. In this data set, however, while the performance of topic based models seems comparable to word based models for smaller lists, as the list size increases, there is a significant improvement in the Recall rate for word based models.

A disadvantage of using topic based techniques is that for LSI and LDA, the number of topics would vary based on the nature and size of the data and it would have to be determined based on the domain. Based on our results, we can say that for duplicate bug detection, word based models are more suitable than topic based ones such as LSI and LDA.

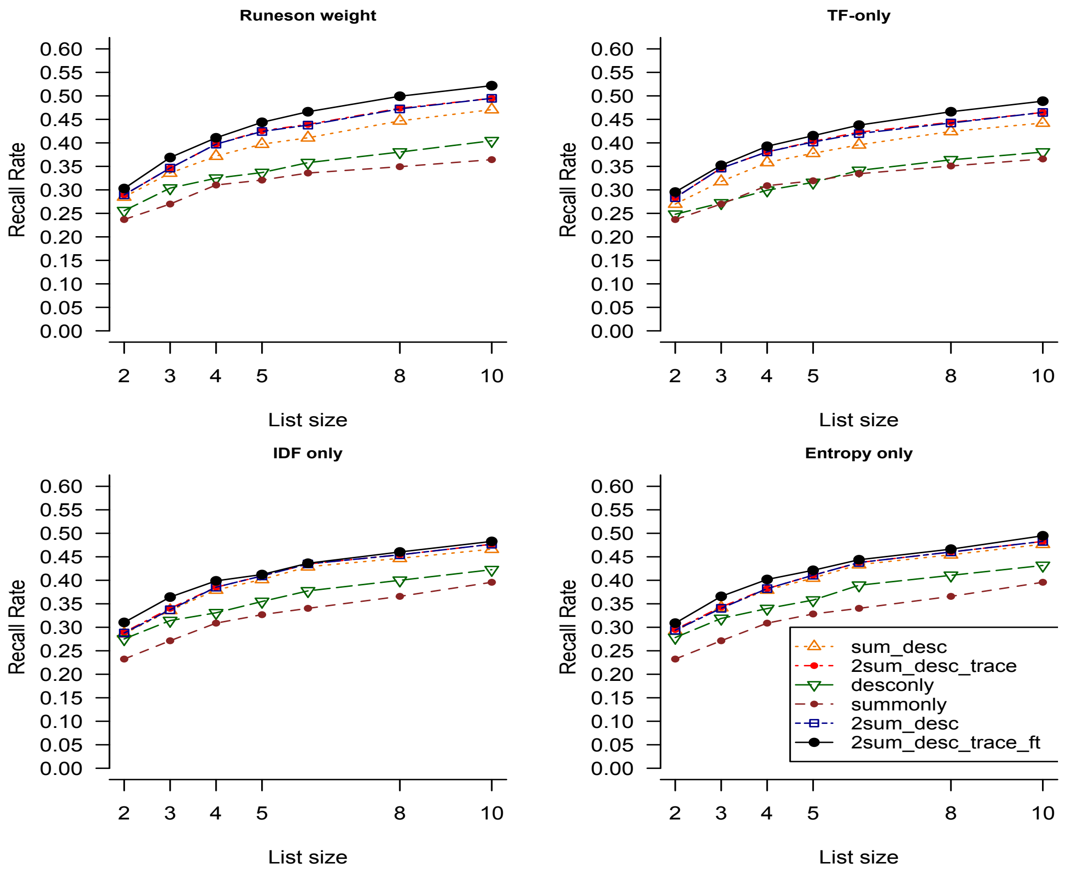


Figure 5.5: Recall Rates on Firefox Using Global and Local Weights Individually (Word Based)

RQ3: *Among word based and topic based models, which ones perform the best in the context of duplicate bug report detection?*

Among topic based models for both data sets, LDA performs worst, with highest Recall rate achieved between 0.12-0.3 for a list size of 2-10 on Eclipse and between 0.2-0.33 on Firefox. Random Projections and LSI seem to be comparable in the Firefox case study, but in Eclipse RP performs slightly better than LSI.

Figure 5.2, shows the Recall rates obtained using local and global weights on the Eclipse dataset. Using a top-list size of 10, the best Recall rate achieved by the logarithmic function of term frequency is 57% , whereas for term frequency it is 53%. Among global weighting

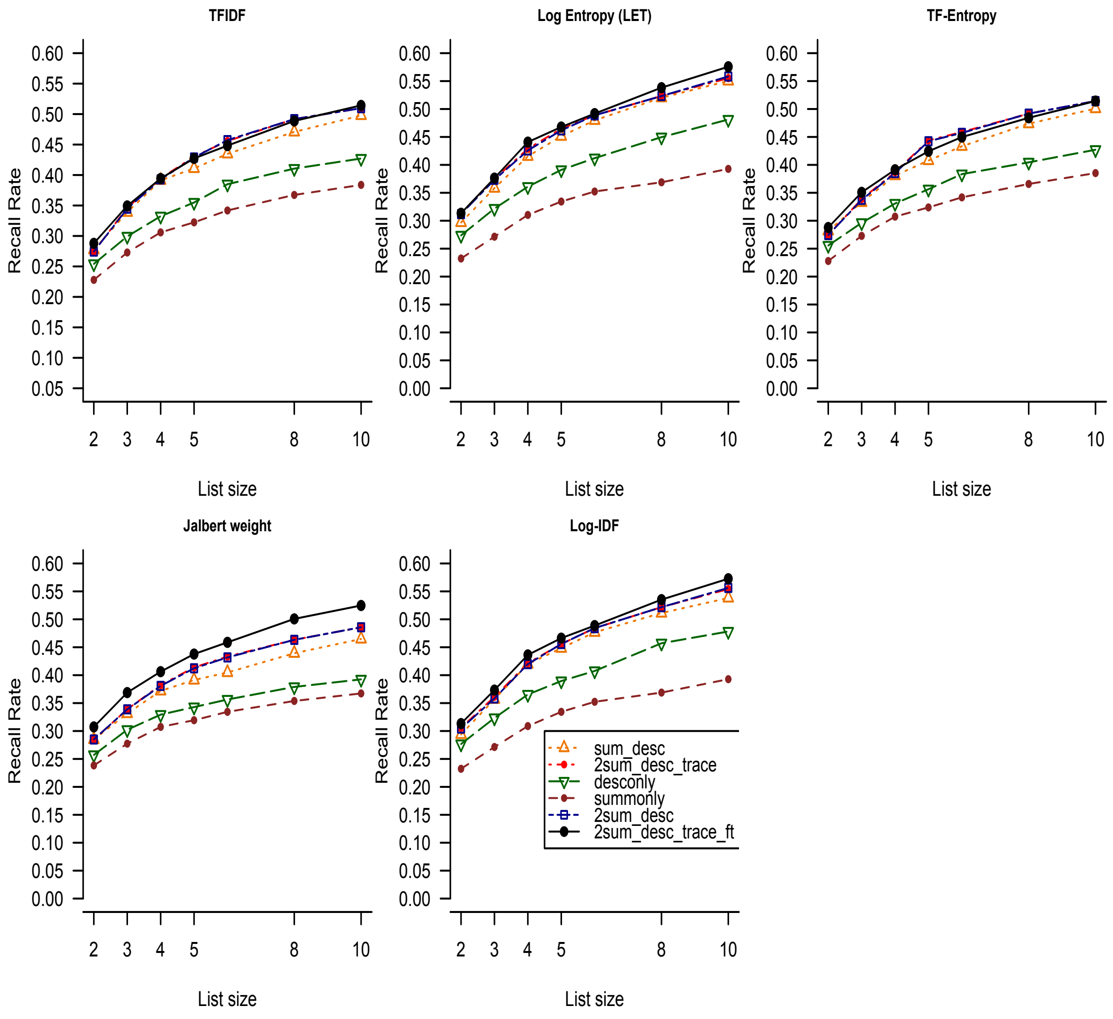


Figure 5.6: Recall Rates on Firefox Using a Combination of Global and Local weights (Word Based)

schemes, the performance of both Entropy and IDF weights is similar. An interesting observation in the case of global weighting schemes is that there is no difference seen in performance by adding a double summary. This is because doubling the word count in a document has no effect on the global weight. Among word based techniques in Eclipse, LET has better performance than the other models on very small list sizes, achieving Recall rates greater than 0.5 on a list of size 5, however, as the list size gets bigger, the other models catch up. On Eclipse, for a list size of 10, a Log-Entropy weight achieves a Recall

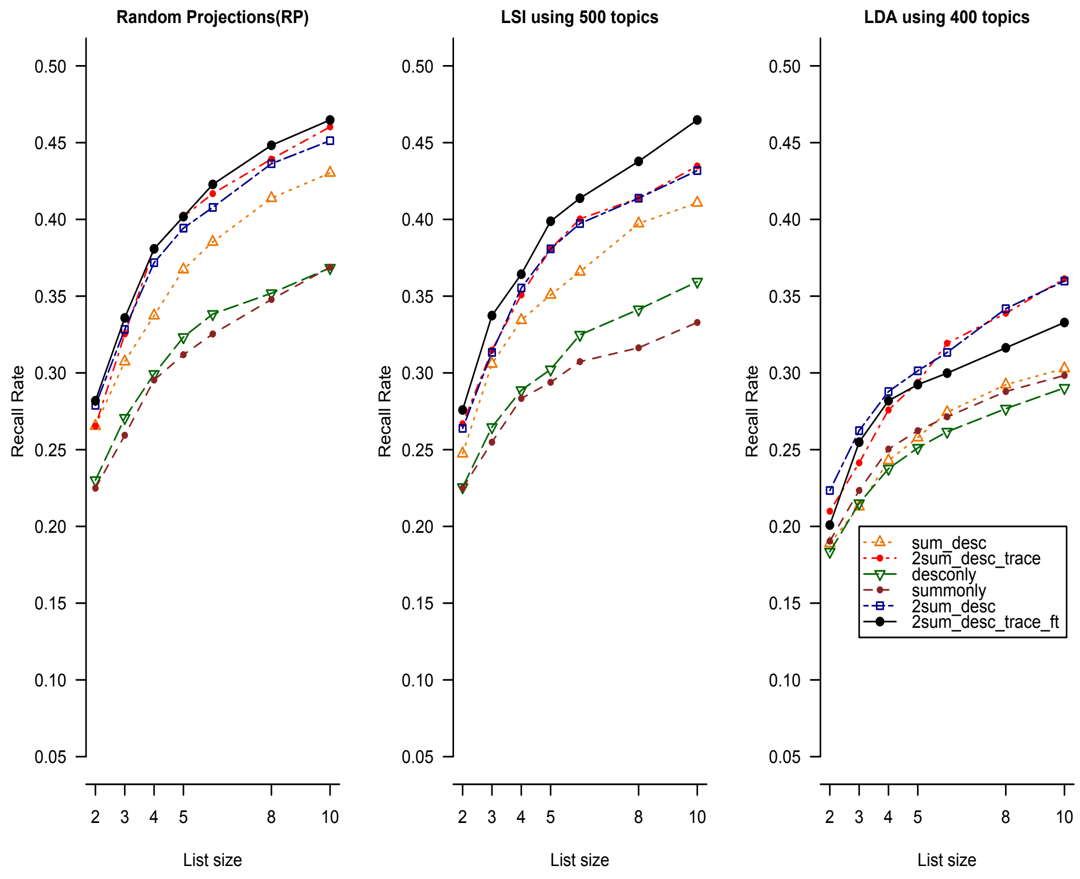


Figure 5.7: Recall Rates on Firefox Using Topic Based Models

rate of 60%, an improvement of 5% over the weighting scheme used in [18], which has lowest performance among word based models.

Regarding the performance of individual global and local weights on Firefox in Figure 5.5, the best Recall rates obtained using a list size of 10 with a logarithmic function of term frequency is 52% while for the raw term-frequency it is 48%. In the case of global weights, both IDF and Entropy weighting schemes have similar Recall rates of 48% and 49% respectively. Looking at the combination of global and local weighting schemes, the Log-Entropy weight achieves a Recall rate of 58%, an improvement of 7% over the TF-IDF weighting scheme which performs worst among the word based mdoels. Therefore, LET is the preferred model among word based techniques, although the performance of the other models is comparable.

Another interesting result is that the use of a double weighted summary only has a marginal improvement over summary and description in most word based models. The improvement is seen the most in the case of weights used by Runeson et al. [39] and Jalbert et al. [18]. This observation is fairly consistent across both the case studies.

While word based retrieval techniques such as TF-IDF tend to be overly specific in matching words in a query to corpus documents, topic based approaches such as LDA and LSI may over-generalize topics for a query. This is evident in the Recall rate of topic based models which is worse than that of word based models. This implies that topic based approaches are less desirable for the problem of duplicate bug detection.

Our results from both the case studies also suggest that in deploying a duplicate bug detection system on a bug repository, it is important to take into account the project's characteristics and determine the heuristics that better suit the application domain in order to achieve best results.

5.2 Online Framework for Duplicate Bug Report Detection

In Section 4.2, the steps to empirically determine which IR models perform better for duplicate bug detection have been described. In order to make a practical retrieval system where duplicate bug detection takes place on the fly, we need to consider three aspects:

1. Knowledge of the best IR model for the data set being used
2. Knowledge of the heuristics that yield best retrieval performance
3. Optimal time frame from which to extract bugs which in turn forms the search space for incoming bugs

We address the first two aspects in our previous case study on Eclipse and Firefox. In Section 4.4, we discuss the significance of time frame in duplicate bug detection that helped to address our fourth research question:
RQ4: *For a required coverage of duplicate detection, what is the optimal time frame for its query space?*

We found that for Eclipse, 70% of duplicate- master bug report pairs could be found between 55 days before and 201 days after the creation of the master bug report. For

NetBeans, 75% of dupliate-master bug report pairs could be found between 3 days before and 190 days after the submission of the master report.

We now introduce an Online Duplicate Bug Report Detection Framework which draws upon findings from the case studies in Section 5.1 as well as knowledge of the amount of duplicate bug report coverage that can be achieved in a given time frame. Figure 5.8 illustrates a block diagram of our proposed online framework for duplicate detection. Assuming we have already indexed bug reports from Y days till Day 1, where Y is the time frame determined from the project's history to maximise the window to find the largest number of duplicates. The following steps outline the steps to realize an online duplicate bug report detection framework:

- Extract the incoming bug report and preprocess the free text through stop word removal, tokenization and stemming to form a bag of words query
- Transform the query in the space of the IR model and index it
- Calculate the similarity between the incoming bug report (query) and the indexed bug reports from the desired "time frame". Obtain a list of the top-N duplicate bug reports.
- Move the query into the set of indexed bug reports and update the index
- Repeat the above for all the incoming bugs for Day 1.
- At the end of Day 1, remove all the bugs that were created Day 1 - Y days ago

In this way, on Day 2, all bug reports from the period of Day 1 - Y days form the search space for all incoming bug reports on Day 2. We repeat the above steps for incoming bug reports on Day 2. By doing so, we maintain a sliding window of bug reports from Y days to form the search space for the incoming bug reports. This is important to ensure that we do not query against either a monolithic or a very minimal set of bug reports.

5.2.1 Data Set

We select a subset of bug reports from Eclipse's Platform project from a period of Jan 2009 till November 2009. Table 5.2 provides the details of the data set. Here, we choose a time frame of 6 months. In other words, bugs from Jan 2009 till June 2009 form the search space for bug reports created from the period of July 2009 till November 2009. Using bug

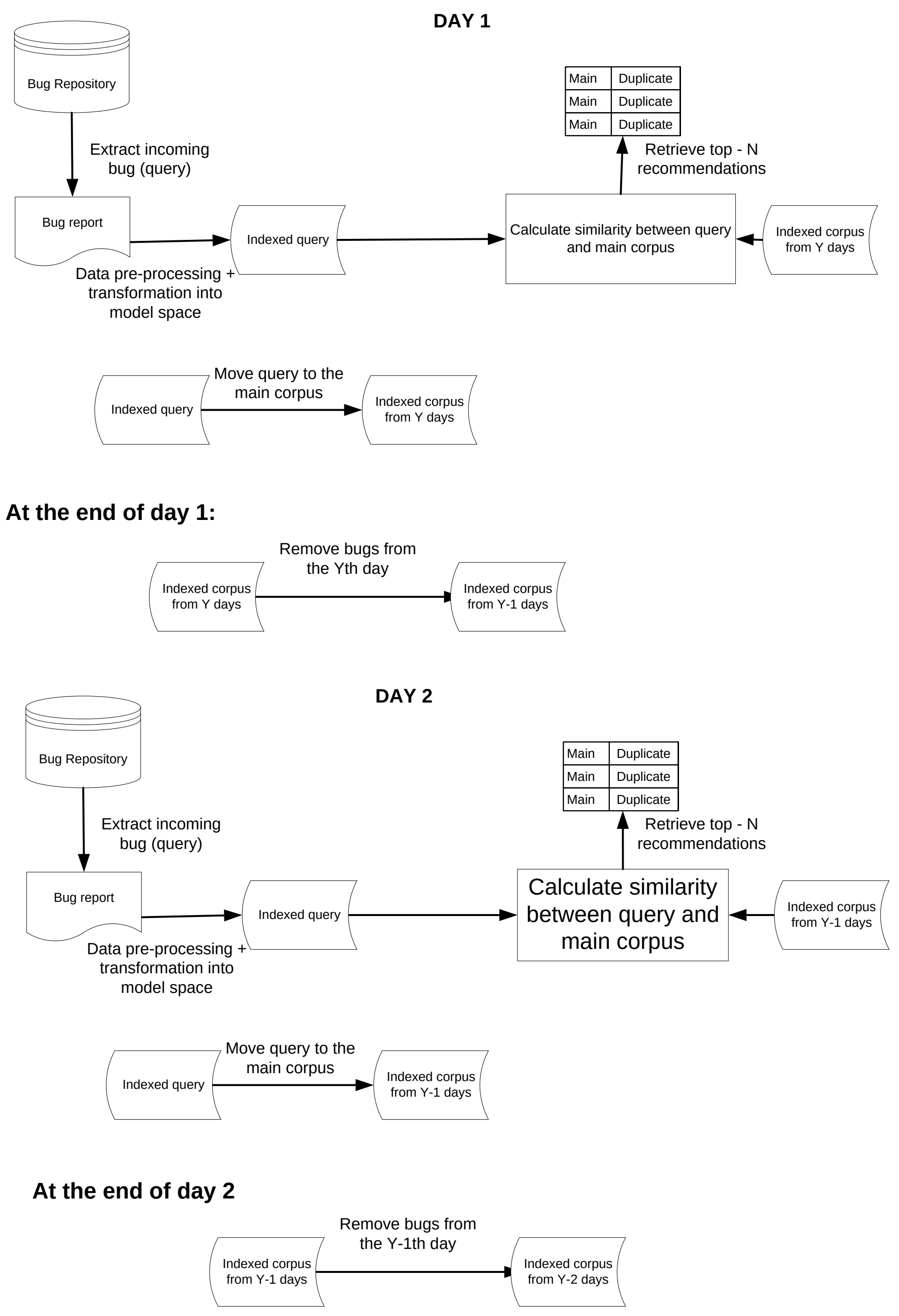

DAY 1
Bug Repository
Extract incoming bug (query)
Bug report
Data pre-processing + transformation into model space
Indexed query
Calculate similarity between query and main corpus
Indexed corpus from Y days
Retrieve top - N recommendations
Main Duplicate
Main Duplicate
Main Duplicate
Indexed query
Move query to the main corpus
Indexed corpus from Y days
At the end of day 1:
Indexed corpus from Y days
Remove bugs from the Yth day
Indexed corpus from Y-1 days
DAY 2
Bug Repository
Extract incoming bug (query)
Bug report
Data pre-processing + transformation into model space
Indexed query
Calculate similarity between query and main corpus
Indexed corpus from Y-1 days
Retrieve top - N recommendations
Main Duplicate
Main Duplicate
Main Duplicate
Indexed query
Move query to the main corpus
Indexed corpus from Y-1 days
At the end of day 2
Indexed corpus from Y-1 days
Remove bugs from the Y-1th day
Indexed corpus from Y-2 days

Figure 5.8: Block Diagram of the Online Duplicate Bug Detection Framework

Description	Value
Project Span of data set	Eclipse (Platform) January 2009 - November 2009
Time Frame	6 months
Initial search space	January 2009 - June 2009
Initial query set	July 2009 - November 2009
Total number of bugs	5537
Total number of duplicates	219

Table 5.2: Dataset for Online Duplicate Bug Report Detection Study on Eclipse

reports from July 2009 till November 2009, we try to simulate incoming bug reports and perform duplicate detection on the fly.

From our results in Section 5.1.4, we determined that for the Eclipse project, a Log-Entropy model performs better than the other models using a heuristic of double weighted summary, the long description and the partial stack frames. For this experiment, we use these heuristics and the Log-Entropy model and record the Recall rate.

5.2.2 Discussions on Results Obtained

A Recall Rate of 57% was achieved with a Log-Entropy model using a double weighted summary, a long description and partial stack frames. This is close to the Recall rate observed in our previous case study on Eclipse where a Recall rate of 60% was achieved.

5.2.3 Benefits of Using the Proposed Framework

The online duplicate bug report detection framework offers a flexible component-based architecture that allows an end user to easily add new IR models and heuristics and use them across different application domains. Due to its architecture, it allows the end user to thoroughly compare the performance of different IR models using desired heuristics. The framework is also customizable for different levels of duplicate bug detection coverage. Determining the time frame for a certain level of coverage potentially decreases the query time as the system does not have to store bug reports from a long period. By decreasing the time frame, one reduces the potential to detect those master and duplicate pairs that were created farther apart in time. Therefore, the onus is on the user to determine a suitable level of duplicate bug detection coverage depending on environmental constraints such as storage. Evidently, with very large, active projects, the rate of incoming bug reports will be high which would mean more bug reports from the past would have to be stored.

5.3 Threats to Validity

Internal Validity: The decision of whether or not to mark a bug report as a duplicate of an existing bug report is subjective and lies on the discretion of the triager. For our study, we assume the triagers' assignment to be correct. Furthermore, results can also be affected by incorrect assignments, missed duplicates and even incorrectly marked duplicate-ids. This is a threat to the internal validity of our experiment. We also assume that the information provided by the bug reporter about the product, component and classification is correct. Reporters may tend to leave this information as generic as possible. Also, sometimes a triager might change the surface features of a bug report if they identify a mis-classification of the bug report.

External Validity: In our case study, we used bugs from the Platform project of the Eclipse bug repository. There may be some characteristics of this project that may not generalize to other projects within Eclipse. This is a threat to external validity.

5.4 Summary

In this chapter, we provide details of the experimental studies. Section 5.1 describes the data sets, experimental setup and evaluation measures used for the two case studies on Eclipse and Firefox. This is followed by a discussion of the results obtained. Section 5.1 addresses the first three research questions. Section 5.2 introduces the online duplicate bug report detection framework and presents results of our preliminary study using the framework on a subset of bug reports from Eclipse. Lastly, Section 5.3 discusses the threats to validity to our studies.

Chapter 6

Conclusion and Future Work

In this chapter, Section 6.1 summarizes the contributions of the thesis and Section 6.2 outlines potential future directions to extend this work.

6.1 Research Contributions

Bug reporting is an essential part of software maintenance in open source software projects. Issue management in large open source projects is done through bug tracking systems such as Bugzilla. Bugs are reported through detailed natural language descriptions of the defects, called bug reports. Due to the large number of bug reports being generated, open source projects hire triagers to help with the task of bug report assignment. One of the issues a triager has to deal with is the identification of duplicate bug reports. Due to the asynchronous nature of bug reporting, the same bug can be reported by several users. Duplicate bug reports, if undetected, can cause duplication of work and effort if the duplicates are handled separately. In the literature, approaches to automate duplicate bug detection have been proposed. However, there has not been an exhaustive comparison of the performance of different IR models, especially topic based ones such as LSI and LDA. In this thesis, we compare the performance word based models having different weighting schemes with that of topic based ones. We define several heuristics and assess their impact on the Recall rate. We carry out our study on subsets of bug reports from Eclipse and Firefox. Based on the findings of our research, we propose an online bug duplicate detection framework which simulates incoming bug reports and provides a list of the top-N duplicates. The major contributions of this thesis can be summarized as follows:

- A comparison of word based and topic based IR models in the context of duplicate bug report detection revealed that word based models outperformed topic based models. We achieved best results using a Log-Entropy based weighting scheme, with a Recall rate of 60% on Eclipse and 58% on Firefox using a top-list size of 10. With topic based models, the best Recall rate was obtained with Random Projections, with a Recall rate of 0.5 on Eclipse and 0.46 on Firefox using a top-list size of 10.

- From the Eclipse data set, we learned that including partial stack frames improved the Recall rate by 5% over using just a double weighted summary and long description. However, using the partial stack frames had no affect on Firefox. In Firefox, the inclusion of surface features improved the Recall rate for some IR models. Our findings suggest that in deploying a duplicate bug detection system, it is important to take into consideration the project's domain and characteristics to devise heuristics accordingly to achieve best results.

- Based on 12 years of historical bugs data from NetBeans, we determined that 70% of master-duplicate bug reports can be found in a time frame of 3 days before and 190 days after the creation of the master report. While for Eclipse, based on 9 years of historical bugs data, we determined that 70% of master-duplicate bug reports can be found in a time frame of 55 days before and 201 days after the creation of the master bug report.

- We realize an online duplicate detection framework that uses a sliding window of a constant time frame to maximise the potential for capturing duplicate bug reports. It is a first step towards simulating incoming bug reports and recommending duplicates to the end user

6.2 Future Work

The research contributions of this thesis pave the way for several possible directions to extend this work. We list them as follows:

- Expand the online duplicate detection framework to include more heuristics and IR models and use these across more application domains.

- Investigate the effects on accuracy of query reformulation and expansion using a thesaurus derived automatically from word co-occurences.

- Incorporate the user of the recommender system in the duplicate retrieval process through relevance feedback. However, for such an approach to be evaluated it is first necessary to implement an online duplicate bug detection system in a production environment.
- Perform a qualitative study investigating the reasons why bug reports are missed by the retrieval system to gain insight for directions to improve the Recall rate.
- Cluster bug reports to generate high-level summaries to help developers save time going through several bug reports with similar content, and ensure that any new information from duplicates is captured.
- Analyze whether the time frame has implications on the quality of retrieval, in other words whether an increase in the time frame increases the number of false positives.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. de Lucia. Identifying the starting impact set of a maintenance request: A case study. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 227–230, February 2000.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering (TSE)*, 28(10):970–983, 2002.

[3] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance (ICSM)*, pages 292–301, 1993.

[4] Atlassian. JIRA. `http://www.atlassian.com/software/jira/overview/`. [Online; accessed November 2011].

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 337–345, 2008.

[6] D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[7] B. W. Boehm and V. R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34:135–137, 2001.

[8] Bugzilla. Most frequently reported bugs. `https://bugzilla.mozilla.org/duplicates.cgi`. [Online; accessed November 2011].

[9] P. H. Carstensen, C. Sørensen, and T. Tuikka. Let's talk about bugs! *Scandinavian Journal of Information Systems*, 7:33–54, 1995.

[10] Y. C. Cavalcanti, E. S. Almeida, C. E. Cunha, D. Lucredio, and S. Meira. An initial study on the bug report duplication problem. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 264–267, 2010.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[12] A. Dekhtyar and J. H. Hayes. Good benchmarks are hard to find: Toward the benchmark for information retrieval applications in software engineering. In *ICSM 2006 Working Session: Information Retrieval Based Approaches in Software Evolution*, 2007.

[13] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods Instruments and Computers*, 23(2):229–236, 1991.

[14] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21:30–36, 1986.

[15] Algorithmic Solutions Software GMBH. About LEDA. http://www.algorithmic-solutions.com/leda/about/index.htm. [Online; accessed November 2011].

[16] E. Greengrass. Information retrieval: A survey. 2000.

[17] T. Hoffman. Unsupervised learning by probabilistic latent semantic analysis. *Machine Learning*, 42(1):177–196, 2001.

[18] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Proceedings of Dependable Systems and Networks (DSN)*, pages 52–61, 2008.

[19] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*, pages 82–85, 2008.

[20] P. Kanerva, J. Kristofersson, and A. Holst. Random indexing of text samples for latent semantic analysis. In *Proceedings of the Annual Conference of the Cognitive Science Society*, volume 1036, pages 103–106, 2000.

[21] Lextek. Stop Word List2. http://www.lextek.com/manuals/onix/stopwords2.html. [Online; accessed November 2011].

[22] M. Lormans and A. V. Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 47–56, 2006.

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

[24] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 125–137, 2003.

[25] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 214–223, 2004.

[26] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[27] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[28] J. N. och Dag. Reqsimile. http://reqsimile.sourceforge.net/. [Online; accessed November 2011].

[29] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. In *Proceedings of the IEEE International Conference on Requirements Engineering*, pages 283–294, 2004.

[30] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. A linguistic-engineering approach to large-scale requirements management. *IEEE Softw.*, 22:32–39, 2005.

[31] J. N. och Dag, T. Thelin, and B. Regnell. An experiment on linguistic tool support for consolidation of requirements from multiple sources in market-driven product development. *Empirical Software Engineering*, 11(2):303–329, 2006.

[32] University of Illinois. NSCA. http://www.ncsa.illinois.edu/Projects/mosaic.html. [Online; accessed November 2011].

[33] B. Pincombe. Comparison of human and latent semantic analysis (lsa) judgments of pairwise document similarities for a news corpus. In *Defence Science and Technology Organisation Research Report DSTORR0278*. DSTO, 2004.

[34] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1997.

[35] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 2002.

[36] V. Rajlich, N. Wilde, M. Buckellew, and H. Page. Software cultures and evolution. *Computer*, 34:24–28, 2001.

[37] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC Workshop on New Challenges for NLP Frameworks*, pages 45–50, 2010.

[38] Mining Software Repositories. Mining Challenge. http://2011.msrconf.org/msr-challenge.html. [Online; accessed November 2011].

[39] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the International conference on Software Engineering (ICSE)*, pages 499–510, 2007.

[40] M. Sahlgren. An introduction to random indexing. In *Methods and Applications of Semantic Indexing Workshop at the International Conference on Terminology and Knowledge Engineering (TKE)*, 2005.

[41] R. J. Sandusky and L. Gasser. Negotiation and the coordination of information and activity in distributed software problem management. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work*, pages 187–196, 2005.

[42] A. Schrter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 118–121, 2010.

[43] Edgewall Software. trac. http://trac.edgewall.org/. [Online; accessed November 2011].

[44] C. Souza, D. Redmiles, G. Mark, J. Penix, and M. Sierhuis. Management of interdependencies in collaborative software development. In *Proceedings of the International Symposium on Empirical Software Engineering (IESE)*, pages 294–, 2003.

[45] C. Sun, D. Lo, X. Wang, J. Jiang, and S. C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 45–54, 2010.

[46] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *Proceedings of the Asia Pacific Software Engineering Conference (ASPEC)*, pages 366–374, 2010.

[47] S. Tangsripairoj and M. H. Samadzadeh. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In *Proceedings of the ACM symposium on Applied computing (SAC)*, pages 1539–1545, 2005.

[48] G. Tassey. The economic impacts of inadequate infrastructure for software testing. In *National Institute of Standards and Technology. Planning Report.* US Department of Commerce, 2002.

[49] Natural Language Toolkit. Natural Language Toolkit. http://www.nltk.org/. [Online; accessed November 2011].

[50] Princeton University. WordNet. http://wordnet.princeton.edu/. [Online; accessed November 2011].

[51] R. Vasile, M. Lintean, and R. Azevedo. Automatic detection of duplicate bug reports using word semantics. https://umdrive.memphis.edu/mclinten/www/Papers/Rus-MSR10.pdf. [Online; accessed December 2011].

[52] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 408–418, 2003.

[53] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 461–470, 2008.

[54] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 513–523, 2002.