2012 16th European Conference on Software Maintenance and Reengineering

A comparative study of the performance of IR models on duplicate bug detection

Nilam Kaushik and Ladan Tahvildari
Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Canada
{*nkaushik, ltahvild*}*@uwaterloo.ca*

***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 capabilities of the bug repository. Both these approaches can be time consuming for the triager and may also lead to the misidentification of duplicates.**

In the literature, several approaches to automate duplicate bug report detection have been proposed. However, there has not been an exhaustive comparison of the performance of different IR models, especially with topic-based ones such as LSI and LDA. In this paper, we compare the performance of the traditional vector space model (using different weighting schemes) with that of topic based models, leveraging 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, LDA and Random Projections. Our findings also suggests that for the problem of duplicate bug detection, it is important to consider a project's domain and characteristics to devise a set of heuristics to achieve optimal results.

***Keywords*-duplicate, bug, information retrieval**

I. 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 [1], with billions of dollars lost due to software defects [2]. As a result, many software projects rely on bug tracking systems to manage their corrective maintenance activities [3].

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 [4]. In effect, a bug tracking system is a hub for users, developers and other stakeholders to engage in the development and maintenance related aspects of the project. In bug tracking systems, issues 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. Due to the large volume of bug reports being created, open source projects incorporate 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. Due to the asynchronous nature of bug reporting in open source software development, the same bugs may be reported multiple times by different users. In order to detect duplicate bug reports, a triager either relies on his memory and experience or on the searching capabilities 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 [3]. 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 [5], [6], [7], [8]. 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.

In this paper, we compare the performance of traditional vector space models with different weighting schemes and topic based models, incorporating heuristics that leverage information such as the summary, long description, exception stack traces and surface features of the bug report. We perform experiments on bug reports from Eclipse and Firefox and achieve a recall rate of 60% and 58% respectively with the optimal set of parameters and a list size of 10. We find that a Log-Entropy based weighting scheme outperforms the other models.

The paper is organized as follows. Section 2 outlines our three research questions and presents two motivating examples that illustrate the kinds of free-text information in the bug report that can be leveraged to detect duplicates. Section 3 presents background on the IR models used in our work and Section 4 discusses the related work. Section 5 details our approach. Section 6 reports on our experimental setup. Section 7 presents our obtained results. Finally, Section 8 concludes the paper.

II. Problem Statement

A. Research Questions

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 [9]. 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. Needless to say, the quality of the suggested list is crucial for the detection of duplicate bugs. Previous works have tried enhancing the quality of suggested lists, but the approaches either suffered a poor recall rate [5], [3] or they incurred additional runtime overhead [7], [6], making the deployment of a retrieval system impractical.

Moreover, till date 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. Dekhtyar et al. [10] claim that Software Engineering is unique in the way IR methods are used in it since the size of document collections in Software Engineering problems tends to be smaller than that of those in standard IR applications. As a consequence, techniques such as LSI that scale poorly on large document collections may yield better results on problems in Software Engineering. In this paper, we address three research questions:

RQ1: *What is the impact of applying different heuristics in helping the detection of duplicate bug reports across the selected case studies?*

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

B. Motivating examples

We present two duplicate bug report pairs from the Eclipse project to illustrate how free-text information in bug reports can be used for duplicate bug detection.

1) Bug-323444 and Bug-330258: In Eclipse, Bug 323444 and 330258 report on a ConcurrentModificationException in the source viewer. Table I summarises information from the summary and long description of the bugs. 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.

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 I: Duplicate Bug-323444 (Product: Platform, Component: UI) and Bug-330258 (Product: Platform, Component: UI)

2) Bug-348680 and Bug-264190: Bug 348680 and Bug 264190 in Eclipse relate to the inability to resize a status bar. 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.
264190	[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 II: Duplicate Bug-348680 (Product: Platform, Component: UI) and Bug-264190 (Product: Platform, Component: UI)

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.

III. IR Models

Information retrieval (IR) is a discipline that aims at extracting useful information from unstructured data, usually expressed in natural language, in response to a query. In this section, we briefly describe two types of approaches- word based, in which each word is assigned a weight as per a weighting scheme, and topic based, the premise being that a document comprises of topics or concepts.

A. Word based

In the motivating examples we observed the usage of common words in the 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 recurrence 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 IR models we use 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 *txd* term-document matrix. The value for a term *i* in a document *j* is *L(i,j) X 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 [11] is the value of the term *i* for the entire collection of documents. In our work, we use these weighting schemes on the Vector space model. We use two local weights- term frequency and a logarithmic function of the term frequency, and two global weights- Entropy and IDF, and we use different combinations of these local and global weights.

1) Term Frequency - Inverse Document Frequency (TF-IDF): In the TF-IDF weighting scheme, the local weight is the term frequency, tf(i,j) and the global weight, idf(i) is the Inverse Document Frequency, given by:

$$log_2(\frac{N_{docs}}{N_i}) \tag{1}$$

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:

$$tf(i,j) * log_2(\frac{N_{docs}}{N_i}) \tag{2}$$

2) Log-Inverse Document Frequency (Log-IDF): In the Log-IDF weighting scheme, the local weight of a term is a logarithmic function of the term frequency given by:

$$log_2(tf(i,j)+1) \tag{3}$$

and its global weight is the Inverse document frequency given in equation 1. The overall term weight is:

$$log_2(tf(i,j)+1) * log_2(\frac{N_{docs}}{N_i}) \tag{4}$$

3) Log-Entropy: In the Log Entropy weighting scheme, the local weight of a term is a logarithmic weight function given by equation 3 and the global weight is an entropy-based function given by:

$$G(i) = 1 - \frac{H(d|i)}{H(d)} \tag{5}$$

where

$$H(d|i) = -\sum_{k=1}^{J} p(i,k) * log_2 p(i,k) \tag{6}$$

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

4) Term Frequency-Entropy(TF-Entropy): In the TF-Entropy weighting scheme, the local weight of a term is the term frequency, tf(i,j) and the global weight is the entropy function given in equation 5. The overall weight of a term is given by:

$$tf(i,j) * G(i) \tag{7}$$

B. Topic based

A bug report 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 such as LSI and LDA expose latent topics, where a latent topic is composed of a group of words that may be used to describe it.

1) Latent Semantic Indexing (LSI): Latent Semantic Indexing (LSI) assumes a latent structure in the usage of words for every document and recognizes topics [12] . 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.

2) Latent Dirichlet Allocation(LDA): Following Probalistic Latent Semantic Indexing (PLSI) [13], a fully generative Bayesian model called Latent Dirichlet Allocation was introduced [14]. 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 [14]. We experiment with a range of topics for both approaches in our study.

3) Random Projections/Random Indexing (RP): Random Projections is an incremental vector space model introduced in [15], 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 [16].

IV. Related work

One of the pioneering works in the area of duplicate bug detection is by Runeson et al [5] which uses Vector Space Models, where each bug report is represented as a vector, and each term in the vector is given a weight of:

$$weight(term) = 1 + log_2(tf(term)) \qquad (8)$$

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. They perform a case study on bug reports from Sony Ericsson. Their results indicate that not all duplicate bug reports can be detected using their approach alone. At best, their approach was able to find 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.

Wang et al.[7] 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:

$$weight(word) = tf(term) * idf(term) \qquad (9)$$

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. With lists of size 1-10, they achieve a recall rate of 67-93% as opposed to 43-72% using NLP techniques only. The drawback of 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. In our work, we use exception stack frames in the long description of bug reports instead of using execution traces.

In [3], 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. They use a different term-weighting scheme for the vector representation:

$$weight(word) = 3 + 2 * log_2(tf(term)) \qquad (10)$$

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.

Rus et al. [8] used the Wordnet lexical database which groups words with similar meanings into synonymous sets. They create an experimental data set from Mozilla's Hot Bugs List and for 20 bugs they retrieved 50 duplicates each, and generate 1000 paris of main and duplicate bug reports, and another set of 1000 non-duplicate pairs. They achieve 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.

In [6], Sun et al. use discriminative models to detect duplicate bug reports. They apply the SVM classifier to classify bug reports into duplicates and non-duplicates and use 54 features to discriminate duplicate bug reports. They use an idf-based weighting scheme and outperform previous approaches that use natural language only by up to 43% on Eclipse, Open Office and Firefox.

V. Proposed approach

A. Heuristics

We retrieve textual information from the bug reports using the heuristics defined below.

1) Double Weighted summary and long description: Runeson et al. [5] suggest that the bug summary should be treated twice as important as the long description. 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.

2) Summary only: In order to see the effectiveness of using terms from the summary, we extract only the summary field from each bug report.

3) Long description only: In order to see the effectiveness of using terms in the long description, we extract only the long description field from each bug report.

4) Equally weighted summary and long description: In this case, we extract both the summary and long description, giving them both equal weight.

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

5) 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 exception stack traces involved changes to one of the stack frames [17]. 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.

6) Surface features: We extract surface features such as the Component, Product and Classification of the bug report.

B. Size of top list

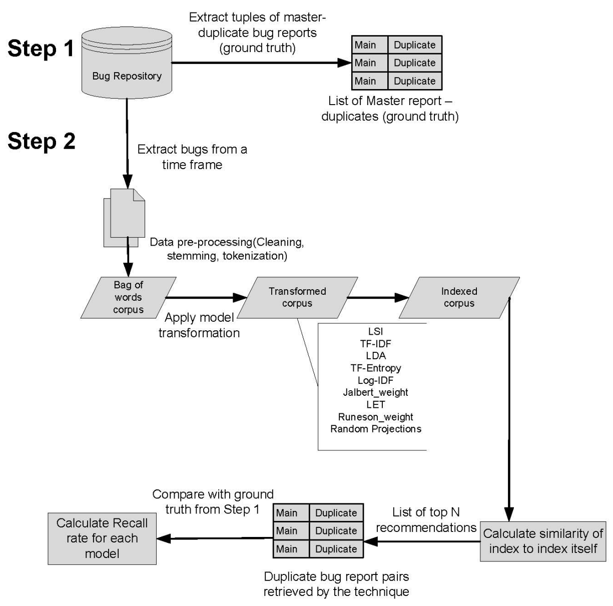
Previous works [5], [7] have shown that using a top list size of 1 is too conservative as the recall rate achieved is poor. Instead, a top list size of 7±2 could be reasonable as Miller [18] suggests in his work on the human capacity for processing information. However, the suitability of this number would have to be experimentally verified when such a system is put into production.

C. Steps to determine 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 2 is a diagram illustrating the steps to determine bug report similarity.

Step 1: For any bugs with the "Resolution" field set to "DUPLICATE", there exists an additional field called "dup id", which is the ID of the master bug report. Therefore, for each bug report in the data set, we check whether its resolution was "Duplicate" and we retrieve the "dup id" and search for it 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 for future reference. This set of all master-duplicate bug report tuples would form our ground truth for the Recall rate calculation in Step 2.

Step 2: From each bug report in the data set, we extract the free-form text depending on the heuristics we are using, and perform some data-preprocessing such as cleaning, stemming, stopword removal and tokenization, as described



Step 1
Bug Repository
Extract tuples of master-duplicate bug reports (ground truth)
Main Duplicate
Main Duplicate
Main Duplicate
List of Master report – duplicates (ground truth)
Step 2
Extract bugs from a time frame
Data pre-processing(Cleaning, stemming, tokenization)
Bag of words corpus
Apply model transformation
Transformed corpus
Indexed corpus
LSI
TF-IDF
LDA
TF-Entropy
Log-IDF
Jalbert_weight
LET
Runeson_weight
Random Projections
Calculate similarity of index to index itself
List of top N recommendations
Main Duplicate
Main Duplicate
Main Duplicate
Duplicate bug report pairs retrieved by the technique
Compare with ground truth from Step 1
Calculate Recall rate for each model

Figure 2: Steps to determine bug report similarity

in Section 6. From the text extracted from all 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 IR 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.

VI. 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.

A. 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 III provides details of the two data sets.

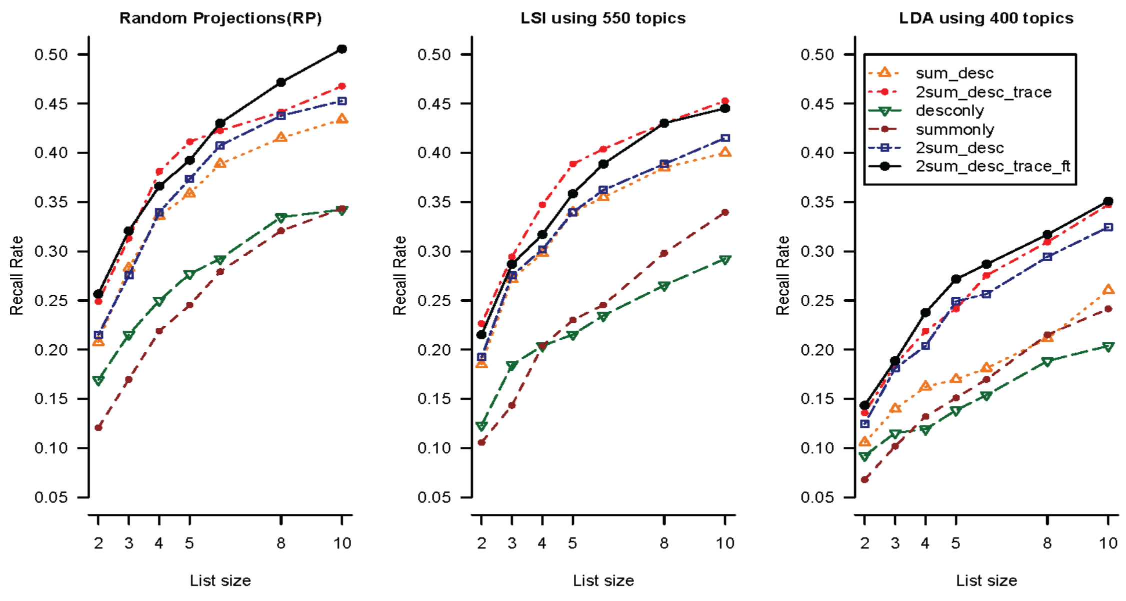


Figure 1: Recall rates on Eclipse using topic-based models

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 III: Summary of 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 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 as “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 the 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 it 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.

B. *Experimental setup*

For the data cleaning and pre-processing steps, we use the Natural Language ToolKit (NLTK)[1], 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[2] 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 sub-

[1]http://www.nltk.org
[2]http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

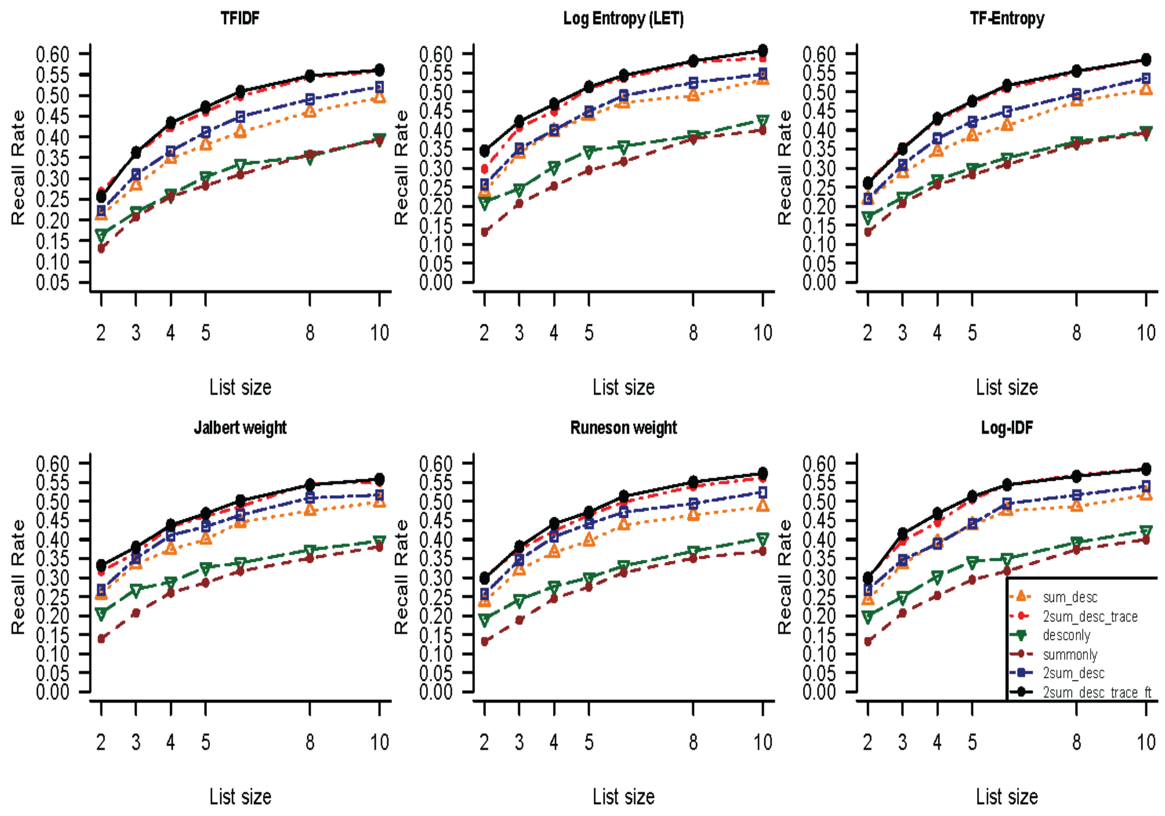


Figure 3: Recall rates on Eclipse using word-based models with different weighting schemes

strings, 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 NLTK.
- Treatment of Exception traces: As per Schroter et al. [17], the top 10 stack frames of bug reports were deemed to be of more importance in bug resolution. 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 [19], a Python framework that extracts semantic topics from documents. While Gensim supports TF-IDF, LET, LSI and LDA, we implement the rest of the weighting schemes to the best of our knowledge upon the Gensim framework.

C. Evaluation measures

We use Recall rate, as defined by Natt och Dag et al. [20] and adapted by Runeson at al. in [5] as a measure of performance.

$$RecallRate = \frac{N_{recalled}}{N_{total}} \tag{11}$$

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 [5].

VII. Obtained results and discussion

In this section, we revisit our three research questions and address them using the results obtained.

RQ1: *What is the impact of applying different heuristics in helping the detection of duplicate bug reports across the selected case studies?*

We discuss the impact of using each of the heuristics defined in Section 5 on the recall rate. Figure 3 and Figure 1 show the results of the word based and topic based models respectively using the Eclipse dataset. Figure 4 and Figure 5 show the results from the word and topic based models on the Firefox dataset. In all the figures, the horizontal axis represents the list size and the vertical axis represents the recall rate.

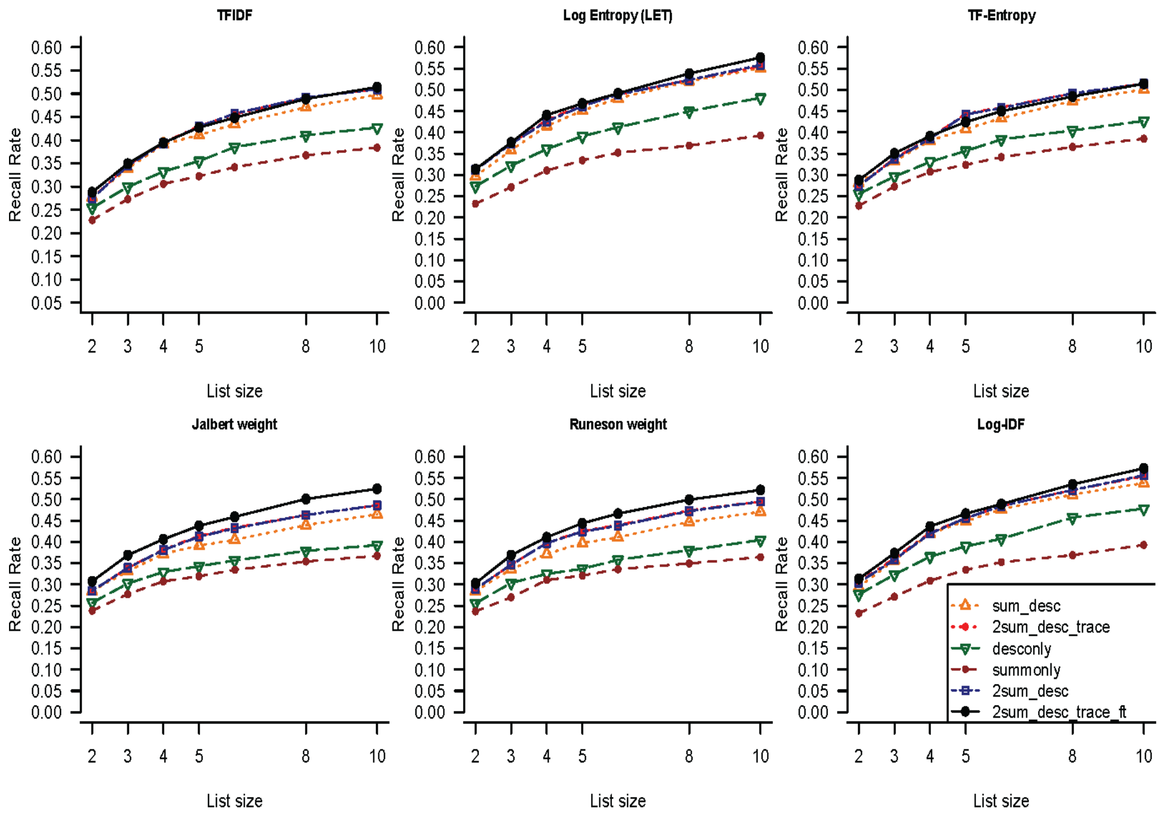


Figure 4: Recall rates on Firefox using word-based models with different weighting schemes

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 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 bug reports in the Firefox dataset tend to 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 for the evaluation.

In both the case studies, word-based techniques outperformed topic-based ones. In Eclipse, 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, for a top size list of 2-10 the Recall rate for topic-based approaches is 0.28-0.46 and for word-based techniques it is 0.31-0.58. In this case, however, while the performance of topic based techniques 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. 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.

Another 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.

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 Random Projections performs slightly better than LSI.

Among word based techniques in Eclipse, LET has better performance than the other weighting schemes 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 rate of 60%, an improvement of 5% over the weighting scheme used in [3], which has lowest performance among word-based models. While for Firefox, 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 models. 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 most evident in the case of weights used by Runeson et al. [5] and Jalbert et al. [3]. This observation is fairly consistent across both the case studies.

While word-based retrieval techniques such as Vector space models 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.

A. Threats to 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. 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.

VIII. Conclusion and future work

In this paper, we have compared the performance of the traditional vector space based model (word based) with that of topic based models on the problem of duplicate bug report detection. We incorporate several heuristics and compare the Recall rates obtained using IR models on bug reports from the repositories of two major open source software projects, Eclipse and Firefox. We find that the vector space model outperforms topic based models and among word based models, a Log-Entropy based weighting scheme has superior performance, achieving Recall rates of 60% and 58% on Eclipse and Firefox respectively. Our study also suggests that it is important to consider a project's domain and charactersitics in order to determine an optimal set of heuristics to be used for duplicate bug detection. As future work, we want to observe the effects of query reformulation and expansion using a thesaurus dervied automatically from word co-occurences. It may also be interesting to involve the user of the bug detection 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. We also plan to perform a qualitative study investigating the reasons why bug reports are missed by our retrieval system to gain insight for directions to improve the Recall rate.

References

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

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," in *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[3] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems." in *DSN'08*, 2008, pp. 52–61.

[4] 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.*, vol. 11, no. 3, pp. 309–346, 2002.

[5] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07, 2007, pp. 499–510.

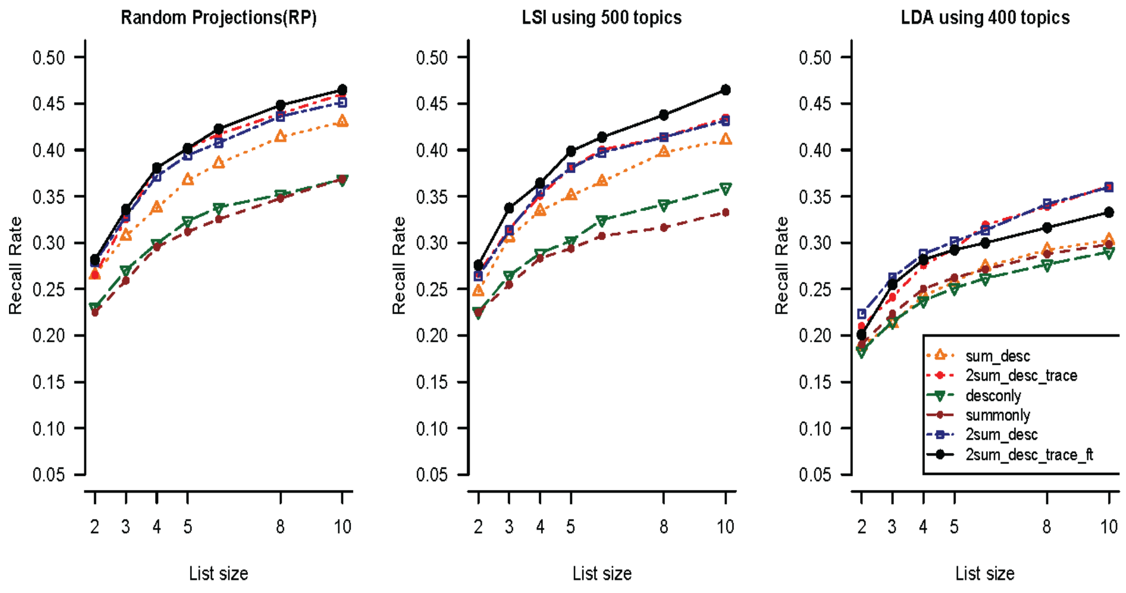


Figure 5: Recall rates on Firefox using topic-based models

[6] 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 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010, pp. 45–54.

[7] 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 *Proc. 30th International Conference on Software Engineering (ICSE 2008)*, May 2008, pp. 461–470.

[8] M. L. Rus. Vasile and R. Azevedo, "Automatic detection of duplicate bug reports using word semantics," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008.

[10] A. Dekhtyar and J. Hayes, "Good benchmarks are hard to find: Toward the benchmark for information retrieval applications in software engineering," 2007.

[11] S. T. Dumais, "Improving the retrieval of information from external sources," *Behavior Research Methods Instruments and Computers*, vol. 23, no. 2, pp. 229–236, 1991.

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

[13] T. Hoffman, in *Machine Learning*, vol. 42(1), 2001, pp. 177–196.

[14] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," in *Journal of Machine Learning Research*, vol. 3, 2003, pp. 993–1022.

[15] P. Kanerva, J. Kristofersson, and A. Holst, "Random indexing of text samples for latent semantic analysis," in *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, vol. 1036, 2000.

[16] M. Sahlgren, "An introduction to random indexing," in *Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005*.

[17] A. Schrter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *MSR'10*, 2010, pp. 118–121.

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

[19] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[20] 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*, vol. 11, no. 2, pp. 303–329, 2006.