Has This Bug Been Reported?

Kaiping Liu
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
kpliu@ntu.edu.sg

Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
ibktan@ntu.edu.sg

Hongyu Zhang
School of Software
Tsinghua University
Beijing, China
hongyu@tsinghua.edu.cn

***Abstract*—Bug reporting is essentially an uncoordinated process. The same bugs could be repeatedly reported because users or testers are unaware of previously reported bugs. As a result, extra time could be spent on bug triaging and fixing. In order to reduce redundant effort, it is important to provide bug reporters with the ability to search for previously reported bugs. The search functions provided by the existing bug tracking systems are using relatively simple ranking functions, which often produce unsatisfactory results. In this paper, we adopt Ranking SVM, a Learning to Rank technique to construct a ranking model for effective bug report search. We also propose to use the knowledge of Wikipedia to discover the semantic relations among words and documents. Given a user query, the constructed ranking model can search for relevant bug reports in a bug tracking system. Unlike related works on duplicate bug report detection, our approach retrieves existing bug reports based on short user queries, before the complete bug report is submitted. We perform evaluations on more than 16,340 Eclipse and Mozilla bug reports. The evaluation results show that the proposed approach can achieve better search results than the existing search functions provided by Bugzilla and Lucene. We believe our work can help users and testers locate potential relevant bug reports more precisely.**

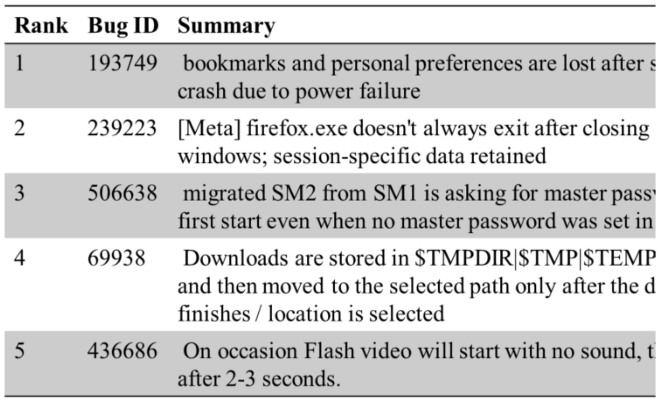
***Index Terms*—Bug report search; Ranking SVM; semantic relation; bug tracking system; search quality**

I. Introduction

Bug tracking systems, such as Bugzilla, have been widely used in software development practice to manage bugs. Users and testers can report problems about a software system via a bug tracking system. The bug reports are stored in the bug tracking system for future reference and tracking. However, the bug reporting process is essentially uncoordinated. Different users and testers may submit bug reports describing the same problem. Previous studies show that as many as 36% of bug reports could be duplicates or invalid [1]. For example, in the Eclipse dataset which includes 316,911 bug reports from 2001 to 2009, 18,124 bug reports have exactly one duplicate and 6,439 bug reports have more than one duplicate. Although some duplicate bug reports could be helpful for developers to debug [2], the existence of duplicate bug reports could still cause extra workload in bug triaging and fixing, and could exacerbate the already high cost of software maintenance.

In order to reduce the redundant effort and maintenance cost, many bug tracking systems require bug reporters to search for previously reported bugs before they submit a new one. For example, the Bugzilla system of Firefox mandates a search step before filing a new bug report [3]. Searching for existing bug reports is also highly recommended by the Bugzilla system of Eclipse.[4]

However, we find that using the existing search functions provided by the bug tracking systems, the same bugs could be still repeatedly reported due to the inadequate quality of the search results. Often, the existing search functions fail to rank the relevant reports high in the output results. Fig. 1 shows an example of the quick search results of Firefox's Bugzilla system (taken on December 10th, 2012). The query is "windows Firefox doesn't release memory long time after page closed".



Rank	Bug ID	Summary
1	193749	bookmarks and personal preferences are lost after s crash due to power failure
2	239223	[Meta] firefox.exe doesn't always exit after closing windows; session-specific data retained
3	506638	migrated SM2 from SM1 is asking for master pass first start even when no master password was set in
4	69938	Downloads are stored in $TMPDIR\|$TMP|$TEMP and then moved to the selected path only after the d finishes / location is selected
5	436686	On occasion Flash video will start with no sound, t after 2-3 seconds.

Fig. 1. An example of search in Firefox's Bugzilla

From Fig. 1, we can see that only five records are returned by Bugzilla given the query. However, after manual examinations, we find that none of these results is actually related to this query. In fact, it is known that such a bug has been reported before [5]. If a user or tester encounters a software problem, but is unable to find a related existing report, it is likely for him/her to file a new but duplicate bug report. The increasing number of duplicate bug reports would require more bug management effort and increase maintenance cost. Therefore, it is desirable to provide a more effective search function to a bug tracking system.

In recent years, many duplicate bug report detection techniques have been proposed [6, 7, 8, 9]. These techniques identify the duplicates among the submitted bug reports, relieving developers of the burden of dealing with these duplicates. Different from these works, our work proposed in this paper focuses on improving the bug report search quality of a bug tracking system. Our work retrieves existing bug reports based on a short user query, before a new bug report is filed. It can help a reporter to locate potential relevant bug reports and determine if she/he would like to continue to report the bug.

In this paper, we propose a novel approach that applies Ranking SVM, a Learning to Rank (LTR) technique, to search for potential relevant bug reports in a bug tracking system. Our approach enables users and testers to locate potential related bug reports more precisely.

In our earlier preliminary work [10], we proposed several features associated with bug reports. In this paper, we enhance the previous work and propose additional features related to a bug report. Especially, we consider the hidden semantic relations among words and documents by using the knowledge of Wikipedia. Latent Semantic Indexing (LSI), which uses singular value decomposition to uncover the latent semantic information among words and documents, is also applied. We perform evaluations using more than 16,340 bug search queries. The results show that our approach can considerably enhance the search quality of the existing bug tracking systems and is better than our previous work. The average F-measure values for the top 10 returned results are around 0.70. Furthermore, experiments are conducted to show which feature combination is more effective for training the ranking model with the aim of further improving the search quality of a bug tracking system.

We summarize our contributions as follows:

- We propose to use Ranking SVM, a state-of-the-art LTR technique to build a ranking model for effective bug report search. Our approach can help a reporter locate potential related bug reports. Based on this, the reporters may decide whether they need to submit a new bug report or not.
- We propose a set of features that can represent the specific characteristics of a bug report. Especially, we consider the semantic features obtained from the analysis of Wikipedia and LSI.
- We conduct experiments on large bug report repositories to evaluate the proposed approach. The results show that our approach outperforms both the conventional search functions and our previous work.

This paper is organized as follows. Section II presents some background information on querying bug reports, ranking models and LSI. Section III describes the features of bug report. Section IV explains the proposed approach. Section V describes our experimental design. Section VI shows the experiment results. Section VII discusses the evaluation results. Section VIII discusses the related work. Section IX concludes the paper and describes the potential future work.

II. Background

A. Querying Bug Reports

In a typical bug tracking system such as Bugzilla, a database is maintained to store bug reports. Table I lists some of the general attributes of a bug report, which include several text fields as well as some categorical and numerical attributes. These attributes constitute the basis for a bug report search engine. According to our observation, similar bug reports have several attributes in common, which implies that users or testers tend to describe the same problem in a similar manner. Hence, these attributes can be utilized to train the bug report retrieval model.

TABLE I. Bug report attributes

Attribute	Description
bug_id	Global ID of the bug report
assigned_to	The developer to whom this bug is assigned
bug_severity	The severity level of the bug report, e.g. Normal, Severe
bug_status	The current processing status, e.g. fixed, unresolved
short_desc	A short description of the bug
long_desc	A detailed description
comments	Comments
op_sys	Affected operating systems
priority	The priority level, e.g. P1~P5
keywords	Keywords describing the bug
product	Name of the product that this bug belongs to
component	Name of the product component that this bug belongs to
version	The CVS/SVN version
reported time	The time stamp when the bug report is submitted
duplicate	Bug IDs that have been marked as duplicates of this bug

B. Conventional Ranking Models

Many conventional ranking algorithms proposed for search engines can be applied to search for existing relevant bug reports given a user query.

Vector Space Model (VSM) is a classic ranking model, in which each term in a document or query set constitutes a dimension of the vector. The relevance between a document and a query can be calculated as the distance between the two vectors, such as the cosine similarity as follows:

$$\cos(\boldsymbol{q},\boldsymbol{d}) = \frac{\sum_{i=1}^{n} q_i . d_i}{\sqrt{\sum_{i=1}^{n}(q_i)^2 \cdot \sum_{i=1}^{n}(d_i)^2}}$$

VSM is intuitive and easy to understand. However, in VSM, there is no consideration of either the order in which the terms appear in the document or the weights of different terms.

LSI is also a commonly-used information retrieval model, which uses Singular Value Decomposition (SVD) to map a high dimensional space into a low dimensional space. In LSI, term-document matrix is used to represent the corpus.

Binary Independent Model (BIM) calculates the relevance between a query and a document as the relevant probability, denoted as $p(R = r|D, Q)$, r = 0, 1. BIM assumes that all terms are independent. Several methods have been proposed to improve the BIM model by addressing the smoothing problem of the unseen terms. A widely-used model is BM25 [11]:

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \times \frac{f(q_i,D).(k_1+1)}{f(q_i,D) + k_1.(1-b+b.\frac{|D|}{agvdl})}$$

where IDF is the inverse document frequency, $f(q_i,D)$ denotes the frequency that the term q_i in query Q appearing in the document D. The k_1 and b are both free parameters for tuning. The *avgdl* is the average document length.

The open source search engine Lucene [12] developed by Apache, uses the following default ranking function:

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t \in q} (tf(t \in d) \times idf(t)^2 \times boost_t \times norm(t,d))$$

This function is in fact a modified VSM model. It calculates the cosine distance between query and document with several normalizations.

C. Learning to Rank Models

A ranking function, which calculates the relevance scores of documents for a given query, is the core of a ranking model. Traditionally, ranking models rely on manual parameter tuning, which is often subjective and lacks generalization ability. In recent years, LTR techniques are proposed to automatically learn an optimized ranking model over large number of features from training data, thus avoiding manual tuning. LTR is mostly to address the ranking of search results [13].

LTR techniques aim at resolving an optimal ranking model for a ranking problem. In a LTR training set, there is a set of query-document pairs <q, d>. For each pair, a set of features can be extracted to form its feature vector. A relevance level, e.g. {perfect, excellent, good, fair, bad}, is assigned to the pair. As such, the ranking model training problem can be transformed to a multi-class classification or regression problem in the n-dimension vector space. The training approaches can be classified into the following major categories [14]:

- Pointwise: in pointwise approach, the training is viewed as an ordinal multi-class classification or regression problem. Thus existing methods such as ordinal SVM and ordinal logistic regression can be applied.
- Listwise: instead of transforming the ranking problem into classification or regression problem, listwise approach such as ListNet [15] aims to optimize the ranking list directly.
- Pairwise: In pairwise approach, the training feature vectors are changed to a binary classification problem. More specifically, for each pair of documents, if they have different labels for one query, the document pair can have two possibilities: the first one's level is better than the second one, or the opposite is true. Hence, the original multi-class problem becomes a binary classification problem.

We adopt the Ranking SVM [16], a pairwise algorithm, which uses SVM to learn a hyperplane that separates the binary classes. Ranking SVM is proved to be effective [16] and has mature open source tool support [17].

III. Features

A crucial part of our approach is the selection of features. We observe that the similar bug reports often share some common attributes, such as operating system, product and component. In addition, there are considerable amount of overlapping words in the short description, comments as well as the long description. These similarities indicate that we can extract features from bug reports for training a ranking model.

A. Textual Features

Term frequency (TF) and inverse document frequency (IDF) are among the mostly used statistical features of texts. They have been used in a variety of ranking functions. Total term frequency (TTF) is the total number of words appearing in a document. We have identified the following textual features based on the concepts of TF IDF and TTF:

- Features only depending on the query, such as the total term frequency (TTF_q).
- Features only depending on the documents, such as total term frequency (TTF_d). In the context of bug report search, more features such as the severity, priority, status, etc. can be included.
- Features of those terms that appear in both a query and a document, such as the term frequency (TF) and inverse document frequency (IDF) values of those terms that appear in both a query and a document.
- The ranking score of BM25 model, which computes similarity scores between a query and the textual descriptions of a bug report.

B. Semantic Features

We exploit two semantic similarity based features. The first one is the similarity score returned by Latent Semantic Index (LSI), which is performed on short description, long description as well as comments. The query vectors are folded into each of the new LSI space of the three fields. The cosine similarities between the new query vectors and the vectors (also in LSI spaces) representing the fields are calculated. Such similarities explore the deep semantic relations among words and documents.

Secondly, we consider the semantic relations among words. Traditionally, semantic models are present in large thesauruses such as Wordnet [18], which mainly captures the shallow, literal relations between words, such as synonyms, hyponyms and antonyms. We believe that in the software domain, deeper semantic relations exist. For example, terms such as "cpu" and "memory" have strong underlying connection although they are not synonyms. With this in mind, it would be better to perform deep mining of semantic relations from a large corpus.

In recent years, Wikipedia has emerged as an enormous knowledge base contributed by nearly 100,000 volunteers around the world. More importantly, articles on Wikipedia are widely accepted as having high quality. In this paper, we adopt the approach proposed in [19] to perform Wikipedia-based semantic analysis. In their work, semantic analysis is treated as link analysis. The relatedness between two articles on Wikipedia is defined by their common in-links and out-links. Specifically, the relatedness measure is defined as follows:

$$\text{relatedness}(\mathrm{a},\mathrm{b}) = \frac{\log(\max(|A|,|B|)) - \log(|A \cap B|)}{\log(|W| - \log(\min(|A|,|B|)))}$$

where *a* and *b* are the two articles of interest, *A* and *B* are the sets of all articles that link to *a* and *b* respectively, and *W* is set of all articles in Wikipedia.

To compare the relatedness of two terms, the terms must be mapped to two articles on Wikipedia so that the relatedness can be computed. However, the disambiguation problem exists since one term may have more than one meanings. To overcome this problem, we adopt the approach used in [20], in which the article having large number of anchor text containing a term is considered as the sense of the term. To adopt this approach, the tool WikipediaMiner [21] is deployed with the Wikipedia database dump of June, 2011. For instance, the term "software" will be linked to the article "Computer Software", while "cpu" will be linked to "Central processing unit". The relatedness between "software" and "cpu" will then be 0.626. If a term cannot be linked to a Wikipedia article, the relatedness between the term and any other terms will be 0.

Using this method, we compute the Wiki-similarity between every two terms from a query and a bug report as the relatedness of their corresponding representative articles. We define the Wiki-similarity as:

$$\text{WikiSim}(q,d) = \frac{\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} relatedness(w_{qi}, w_{dj})}{|q| + |d|}$$

where w_{1q} and w_{2d} are terms from the query *q* and bug report *d*. The result of the function WikiSim is essentially the average of relatedness computed for each pair of words from the query and the bug report. This result is also treated as a feature of short descriptions and keywords.

C. Categorical and Numerical Features

We also use some bug-related information such as report time, operation system, bug status, and priority in our work. All these information can be extracted as features of bug reports. The status and priority are transformed into discrete values. The bug report time is transformed to the standard UNIX time (as the seconds since 1970-1-1, 00:00) and then normalized into real values in [0, 1].

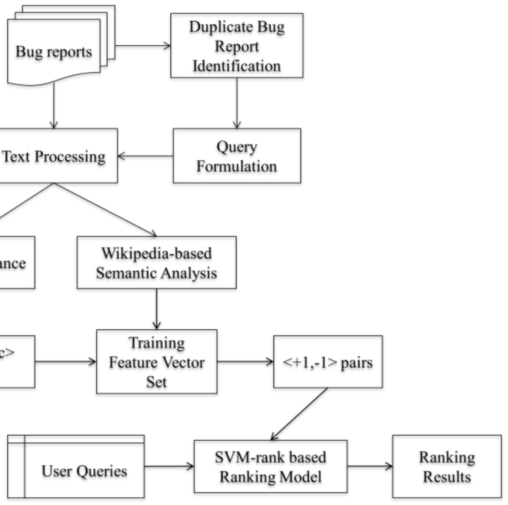
D. Feature Vector

For a bug report, we extract in total 83 features as shown in Table II. The minimum, maximum, sum, mean and variance of the TF, IDF and TF * IDF values are chosen as the features to train our model, together with the BM25 score, LSI and the WikiSim.

These values are computed using both a query and a variety of bug report fields, including short description, long description, comments, and keywords. As an example, the sum of TF is defined as follows:

$$\sum_{t \in q \cap d} f(t,d)$$

in which *q* denotes the query, and *d* denotes the bug report. *f(t, d)* denotes the number of times that *t* appears in *d*.



Bug reports
Duplicate Bug Report Identification
Text Processing
Query Formulation
Label Relevance
Wikipedia-based Semantic Analysis
<query, doc> label
Training Feature Vector Set
<+1,-1> pairs
User Queries
SVM-rank based Ranking Model
Ranking Results

Fig. 2. Overview of the bug report search framework

TABLE II. FEATURES USED TO TRAIN RANKING SVM MODEL

Category	Attribute	Feature	No. of Features
Textual	query	TTF_q	1
	short_desc	TTF_d	17
		Sum, Min, Max, Mean and Standard Variation of TF, IDF, and TF*IDF	
		BM25	
	long_desc	TTF_d	18
		Sum, Min, Max, Mean and Standard Variation of TF, IDF, and TF*IDF	
		WikiSim	
		BM25	
	comments	TTF_d	17
		Sum, Min, Max, Mean and Standard Variation of TF, IDF, and TF*IDF	
		BM25	
	keywords	TTF_d	17
		Sum, Min, Max, Mean and Standard Variation of TF, IDF, and TF*IDF	
		BM25	
Categorical and Numerical	op_sys	TF_{sum}	1
	product	TF_{sum}	1
	component	TF_{sum}	1
	version	TF_{sum}	1
	bug_status	Discrete Value	1
	priority	Discrete Value	1
	bug_severity	Discrete Value	1
	reported time	Normalized to [0, 1]	1
Semantic	short_desc	WikiSim	1
		LSI	1
	keywords	WikiSim	1
		LSI	1
	comments	LSI	1
	Total No. of Features: 83		

IV. The Proposed Approach to Bug Report Search

Fig. 2 shows the overview of the proposed bug report search framework. A training set based on existing bug reports is first constructed. When the queries, bug reports, feature vectors and labels are obtained, the ranking model can be trained. The trained model can then be used for retrieving bug reports based on a user query. This section describes these steps in detail.

A. Constructing Training Set

The process of constructing a training set from the existing bug reports contains the following major steps:

1) Identifying Duplicate Bug Reports: The bug reports that are marked as duplicate are grouped together to form a duplicate group. Following the definition in [6], the earliest submitted bug report in one duplicate group is marked as the master bug report.

2) Query Formulation: In the training process, for building the model, the short description and keywords of the master bug report are combined to form one query. It is worth noting that the reason we only choose short description and keywords as query is that we aim at improving the search quality of bug tracking system. Therefore unlike previous work on duplicate bug report detection, we do not make use of the complete bug report.

3) Relevance Label Formulation: We adopt five levels of relevance grade: {Perfect, Excellent, Good, Fair, Bad}. First of all, assume that for a duplicate group, there are k duplicate bug reports excluding the master report. These k bug reports are labeled as "Perfect". To ensure there is an adequate number of high relevance bug report for a query, we set a lower bound α and an upper bound β for k, i.e. $\alpha \leq k \leq \beta$. We label at most N bug reports in total for each query where $N > \beta$ in order to balance the training dataset by mixing both the high and low quality bug reports. However, we do not have labeled training data for our ranking model. Meanwhile, it is difficult to label the training dataset manually since there are a large number of bug reports. To obtain bug reports with other relevance grades, we make use of the search results of Bugzilla's built-in quick search. For a query that has k ($\alpha \leq k \leq \beta$) associated duplicate reports, the top N-k Bugzilla search results excluding the duplicates are considered. For these N-k bug reports, the top 25% are marked as Excellent, the next 25% as Good, the subsequent 25% as Fair and the last 25% as Bad. Here, in order to balance the training set, we make the remaining{Excellent, Good, Fair, Bad} grade equally distributed in the training set. Experiment is conducted to justify the labeling approach. The evaluation result is shown in the section VII.A.

4) Feature Vector Generation: The queries and textual bug reports require the following preprocessing steps:

- Word regularization: All letters are lowercased. Words are segmented by standard whitespace characters.
- Word stemming: Stemming is a commonly used technique to normalize words with the same root by removing their suffixes. For example, compute is the stem of computes, computing and computed. We use Snowball [22], a widely used stemming library in our approach.
- Word splitting. In bug reports, there is much source code mixed in the text. We follow [23] to split the symbols of functions and variables in the source code.
- Stop words removal: Stop words are those words that do not contribute any actual semantic meaning, such as the propositions. We remove the stop words according to the stop word list provided by MIT [24].

After preprocessing is done, all features of each query-bug report pair can be computed.

B. Constructing the Ranking Model

We apply Ranking SVM [16] to train the ranking model. Ranking SVM is a pairwise LTR algorithm, in which the hyperplane is learned over pairs of feature vectors with different relevance labels.

To obtain the input for training the Ranking SVM model, the training dataset created in the previous section should be converted into feature vector pairs. For example, assume that for a query *Q*, its first 5 bug reports have the following relevance grade: *{Perfect, Excellent, Good, Good, Fair}*. The feature vectors of these 5 bug reports are x1 ~ x5, respectively. We could obtain the pairs and their labels as follows:

$[<\mathbf{x}_1, \mathbf{x}_3>, +1], [<\mathbf{x}_3, \mathbf{x}_1>, -1],$

$[<\mathbf{x}_1, \mathbf{x}_4>, +1], [<\mathbf{x}_4, \mathbf{x}_1>, -1],$

$[<\mathbf{x}_1, \mathbf{x}_5>, +1], [<\mathbf{x}_5, \mathbf{x}_1>, -1],$

……

$[<\mathbf{x}_3, \mathbf{x}_5>, +1], [<\mathbf{x}_5, \mathbf{x}_3>, -1]$

For each training feature vector pair $<\mathbf{x}_i, \mathbf{x}_j>$ where $\mathbf{x}_i$ and $\mathbf{x}_j$ are associated different relevance, if the label of $\mathbf{x}_i$ is better than the label of $\mathbf{x}_j$, then the pair $< \mathbf{x}_i, \mathbf{x}_j >$ will be labeled with +1, otherwise, $< \mathbf{x}_i, \mathbf{x}_j >$ will be labeled as -1. Hence, the original multi-class problem is transformed into a binary classification problem and an SVM classifier can be trained over the binary-labeled pairs. The transformed pairs are shown in Fig. 3 [14].

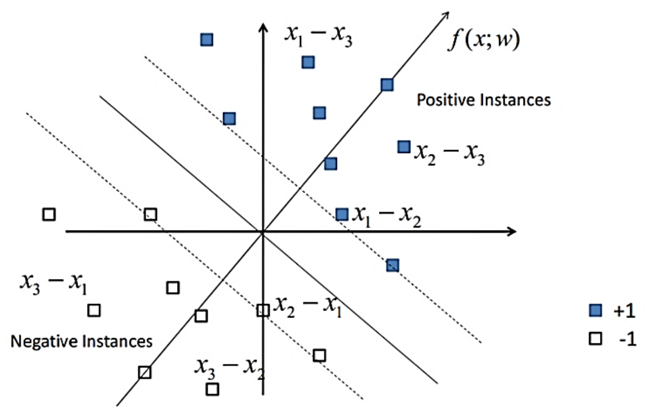


Fig. 3. Pairwise approach for LTR (Ranking SVM)

In Fig. 3, the shaded points denote the ordered pairs that the first feature vector's label is better than the second one's, and the white points denote the opposite cases. The SVM learning algorithm tries to learn an optimal hyperplane w that separates these two types of points with largest margins.

In Ranking SVM, the learning process is to optimize the loss function with quadratic programming:

$$min_{w,\xi}\frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i$$

$$s.t.\, y_i\,\langle w, x_i^{(1)} - x_i^{(2)}\rangle \geq 1 - \xi_i$$

$$\xi_i \geq 0, i = 1 \ldots, m$$

in which m is the number of training instances and $xi^{(1)}$ and $xi^{(2)}$ are the i^{th} pair of feature vectors. w is the hyperplane which we try to optimize and ξ are the slack variables which allow some points cross the hyperplane.

C. Use of the Ranking Model

For a query q and two bug reports d_i, d_j, two 83-dimension feature vectors X_i, X_j can be obtained. Then the ranking order between them can be calculated as:

$$y = \boldsymbol{w} \cdot (\boldsymbol{X_i} - \boldsymbol{X_j})$$

in which w is the hyperplane learned from the training data. If $y > 0$, d_i should be ranked higher than d_j, and vice versa.

V. Experimental Design

A. Data Collection

We collected data from two bug report repositories from the Bugzilla databases of Eclipse and Mozilla, both of which are popular, large-scale open source systems. The Eclipse dataset consists of 316,911 bug reports in total collected from 2001 to 2009. The Mozilla database has 572,400 bug reports collected from 2001 to 2008. These datasets are stored in a MySQL database dump file, which consumes about 1.7GB of disk space.

There could be a number of duplicate bug reports in a duplicate group. The highest number of duplicates that one bug report has is 49. Considering the time efficiency, in our experiments, we only chose duplicate groups that contain at least 2 and at most 10 bug reports, i.e. $\alpha = 2$ and $\beta = 10$. There are 6,340 such duplicate groups in total in Eclipse's database. Since Mozilla's database is very large, we randomly selected 10,000 such duplicate groups from it. Since the master bug report of each duplicate group is used to form the query, the queries totaled to 16,340. Next, we randomly selected 20% of the chosen groups from each dataset (1,268 from Eclipse and 2,000 from Mozilla) to form the training set, and the rest are used as testing dataset. When we applied the Bugzilla's built-in quick search for each query, we set the number of documents to be returned to 20, i.e. $N = 20$. Therefore, for each query, at most 20 feature vectors are generated for the returned documents.

Since N is set to be 20 in our experiment, when the pairwise approach is applied, the number of pairs would be very large. To further reduce the training size, we randomly sampled 50,000 pairs from Eclipse dataset and 100,000 pairs from Mozilla dataset and made the training pairs equally distributed with +1 and -1 labels. After the ranking models are trained, the queries of the testing set are run on the datasets and the search results are evaluated against the labels. Table III shows the statistics of the data used in our experiments.

The training and testing were conducted by making use of SVM^{Rank} [17], a tool implementing the Ranking SVM algorithm. The trained Ranking SVM based model can then be used for retrieving relevant bug reports based on a short user query. An experiment was conducted by using the datasets described above. All the experiments are performed on a server with an eight-core Intel Xeon 2.9GHz CPU and 8GB memory.

TABLE III. Statistics of Eclipse and Mozilla datasets

	#Total Bug Report Duplicate Group	#Training Groups	# Training Pairs	#Testing Groups
Eclipse	6,340	1,268	50,000	5,072
Firefox	10,000	2,000	100,000	80,000
Total	16,340	3,268	150,000	85,072

B. Research Questions

We want to answer the following research questions:

RQ1: Is the proposed approach more effective than the conventional bug report search methods?

To answer RQ1, we compared the evaluation results of the proposed approach with those of the default search engine of Bugzilla and those of the Lucene search engine. The quick search function provided by Bugzilla uses a Boolean model which connects all query words with the logical connecter "OR". Lucene uses a VSM model which calculates the normalized TF/IDF score. For Bugzilla, the Eclipse and Mozilla databases are imported into MySQL. We used a Python script to submit the queries by using the HTTP forms of the quick search of Bugzilla. For Lucene, the full text of short descriptions, long descriptions and comments are indexed. The queries of the testing dataset are executed by using the API provided by Lucene.

RQ2: What is/are the more effective feature category/categories for training the ranking model?

To answer RQ2, we designed experiments by using different feature categories and compared their effectiveness in training the Ranking SVM model. The 83 features described in Section III can be further divided into three groups (categories): G1: textual features; G2: semantic features; G3: categorical and numerical features.

We conducted three experiments to answer RQ2. The first experiment only used features in G1, the second experiment used features in both G1 and G2, and the third experiment used features in all G1, G2 and G3. We compared the performances among the experiments and inspected the weights of the features in the ranking models.

RQ3: What is the time efficiency of the proposed approach?

In RQ3, we examined the efficiency of the proposed approach. We measured the training time against different sizes of the training set, i.e. the number of pairs used for training the Ranking SVM models. We would like to find out whether the time performance of the proposed approach is acceptable to real-world bug report search scenarios.

C. *Assessing Performance*

Since our approach aims to produce a ranking which favors the existing related bug reports, we measured the average number of duplicates that exist above the ranking positions 1, 3 and 5 (N@1, N@3, and N@5) of all the queries in the testing dataset.

The Mean Average Precision (MAP), which is the mean value of the average of relevance score of all queries, is defined as follows:

$$MAP(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{m_j} Precision\ (R_{jk})$$

where Q is a set of queries, R_{jk} is the ranking results of one query and m_j=10. The values of MAP are between 0 and 1, the higher the values, the better the retrieval performance.

F-measure combines both precision and recall which makes it useful as there is a tradeoff between precision and recall. The average F-measure at position 10 (F@10), is calculated as:

$$F@10 = \frac{2 \times P@10 \times R@10}{P@10 + R@10}$$

where average *P@10* and average *R@10* are precision and recall at position 10 respectively.

VI. EXPERIMENTAL RESULTS

A. *Results of RQ1*

To answer RQ1, we compared the evaluation results of the proposed approach with the results of the default search engine of Bugzilla and Lucene.

As shown in Table IV, Ranking SVM outperforms the methods used in both Bugzilla and Lucene on all the other metrics. For example, for the Eclipse dataset, the N@3 value achieved by Ranking SVM is 23.5% higher that the value obtained by Bugzilla, and 28.6% higher than the value obtained by Lucene. Furthermore, the MAP of Ranking SVM is 41.8% higher than the value of Bugzilla and 56.4% higher than that of Lucene.

For Mozilla dataset, in comparison with Bugzilla, Ranking SVM also improves with 29.3% for the measurement N@3 and 22.6% for the measurement F@10. The comparison results show that by employing Ranking SVM, users and testers are able to locate previously submitted related bug reports from the search results more effectively. After viewing all related bug reports, the users and testers may decide whether they need to submit a new bug report.

TABLE IV. THE EXPERIMENTAL RESULTS OF BUG REPORT SEARCH METHODS

Eclipse	**N@1**	**N@3**	**N@5**	**MAP**	**F@10**
Ranking SVM	0.75	1.26	2.25	0.61	0.83
Bugzilla	0.63	1.02	1.88	0.43	0.70
Lucene	0.61	0.98	1.64	0.39	0.69
Mozilla	**N@1**	**N@3**	**N@5**	**MAP**	**F@10**
Ranking SVM	0.64	1.06	2.37	0.65	0.65
Bugzilla	0.55	0.82	1.56	0.59	0.53
Lucene	0.52	0.85	1.65	0.55	0.47

B. *Results of RQ2*

To answer RQ2, we conducted three groups of experiments as mentioned in the section V.B. All these three experiments were conducted on both Eclipse and Mozilla. The trained ranking models were applied on the two testing datasets (shown in table III) respectively. In this way, we eventually obtained 6 groups of evaluation results.

The evaluation metrics, including the N@1, N@3, N@5 and MAP, are computed for the three groups of feature combinations on both Eclipse and Mozilla testing sets. The results are shown in Table V. For the Eclipse dataset, the feature combination G1+G2 outperforms G1 by 4.62% on N@1, and 2.69% on N@5. The feature combination G1+G2+G3 further surpasses G1+G2 for N@3 by 36.96%.

For the Mozilla dataset, it can also be observed that G1+G2 achieves better performances on all the measurements besides F-measure than G1. For example, N@1 is improved by 5.26% and N@5 by 3.62%. G1+G2+G3 gains notable advantages over all the measurements of both G1 and G1+G2. For example, compared to G1+G2, N@3 is improved by 13.98%.

In Table VI, the trained weights, i.e. the corresponding values in the hyperplane vector ***w*** are shown for the top 10 features with the highest weights.

From Table VI, it can be seen that the feature short_desc TF*IDFmean has the highest weight, which means that this feature is the most distinguishing one among the 83 features. The weight information reveals which features are more useful in the search of existing related bug reports. Such information can guide users or testers to submit more useful and precise information so as to improve the overall search quality.

TABLE V. THE COMPARISIONS OF FEATURE COMBINATIONS

Eclipse	**N@1**	**N@3**	**N@5**	**MAP**	**F@10**
G1	0.65	0.97	1.86	0.62	0.64
G1+G2	0.68	0.92	1.91	0.60	0.72
G1+G2+G3	0.75	1.26	2.25	0.61	0.83
Mozilla	**N@1**	**N@3**	**N@5**	**MAP**	**F@10**
G1	0.57	0.90	2.21	0.63	0.60
G1+G2	0.60	0.93	2.29	0.59	0.63
G1+G2+G3	0.64	1.06	2.37	0.65	0.65

TABLE VI. TOP 10 FEATURES WITH HIGHEST WEIGHTS

Feature	Weight	Feature	Weight
short_desc TF*IDFmean	0.374	short_desc TF*IDFmean	0.128
short_desc BM25	0.302	long_desc IDFsum	0.115
long_desc TFmean	0.261	comments TF*IDFsum	0.106
comments TFmean	0.174	long_desc IDF-max	0.091
op_sys TFsum	0.159	comments IDFmax	0.073

In summary, the results show that features in G1 are essential for bug report search. Features in G2 and G3 capture more specific information of a bug report including the semantic similarities and categorical and numerical attributes, and are helpful for reducing ambiguity and improving performance.

C. *Results of RQ3*

To answer RQ3, we measured the training time against different sizes of the training set. Fig. 4 shows the time efficiency results for training process.

In Fig. 4, the horizontal axis represents the number of training pairs and the vertical axis represents the training time in seconds. When there are 30,000 training pairs, the training process can be finished in about 2.5 minutes. When there are totaled 100,000 training pairs, the training process is finished in about 14 minutes. From Fig. 4 it can be also seen that the training time scales nearly linear with regard to the training set size. To measure the average time for answering a query, 1,000 queries are generated on a bug report size 146,860, the average time for answering a query is 73.7ms with standard deviation of 15.4. In summary, we believe that the time efficiency of the proposed approach is acceptable for the real-world bug report search scenarios.

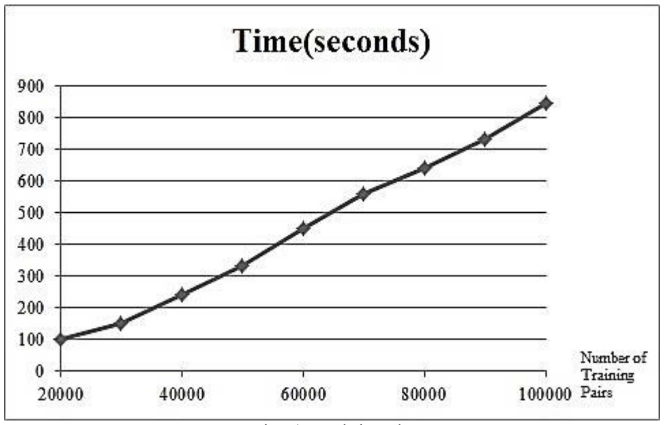


Fig. 4. Training time

VII. DISCUSSION OF THE RESULTS

A. *Discussions*

From the evaluations, it can be seen that the proposed approach outperforms the built-in search function of Bugzilla as well as the classic Lucene search engine. The reasons behind this are as follows:

- The selected features cover the textual characteristics of various fields of a query and a bug report. Furthermore, the other features such as bug priority provide additional useful information for measuring the similarity between a query and a bug report, and improve the effectiveness of the ranking model.
- The pairwise LTR method learns an optimized weighting model over the set of features and hence manual parameter tuning is avoided.
- Semantic relations among words and documents are explored. We considered the semantic features obtained from the analysis of Wikipedia and LSI.

As mentioned in Section IV, we choose *N* bug reports for each query and *N* is an arbitrary positive integer larger than *k*. We conducted experiments with different *N* values (20, 30 and 40). We discovered that there was no significant differences in results when $N = 30$ and 40. It is because a larger *N* value for a query would include these bug reports with lower relevance, thus the quality of the training set would be compromised.

Since we do not have labeled dataset for training bug report search ranking model, and our resources are also too limited to create a manually labeled dataset, we decided to use the search results of Bugzilla for the labels other than "good". To justify this approach, we randomly sampled 50 queries with their labeled bug reports from the Mozilla training set. Manual inspection was conducted and the manual labels for the bug report for each of the 50 queries were created. The manual labels and the labels from the training dataset were compared using Wilcoxon signed-rank test and the results calculated were $N_r = 312$ and $W = 574$. Hence the null hypothesis was accepted with $z = 0.179$ and $\alpha = 0.05$ ($z < z_\alpha = 1.96$).

Although our experiments show that the proposed approach can effectively retrieve relevant bug reports, it may still fail to rank some existing related bug reports higher (e.g., in top 10). Our model assumes that duplicate bug reports have high degree of common text information, which is a valid assumption for most of the duplicate bug reports. However, we found that this assumption not always holds due to synonyms and hyponyms. In this paper, we have explored semantic analysis by applying Wikipedia link analysis and LSI. In the future, we plan to incorporate more nature language processing techniques to further improve the semantic analysis of bug reports.

B. *Threats to Validity*

There are potential threats to the validity of our work. Like other data mining research work, our conclusions could be biased by the type of data that were used. To validate our approach, we perform evaluations on more than 16,340 bug reports collected from two popular, large-scale open source systems. However, the datasets we used in our experiment could still be small and biased. In the future, we will extend our experiments to more projects, including both open source and industrial projects.

The dataset could also contain label bias. In our labeling process, we make use of the quick search results of Bugzilla, which may not be fully accurate. However, we used Wilcoxon signed-rank test to evaluate the labeling method. The statistical hypothesis testing accepts the null hypothesis. Hence, we believe the labels given based on the Bugzilla's quick search are accurate. Last but not least, the proposed approach depends on the accuracy of queries entered by the users. If the user query is ambiguous, the results could be under threat.

Furthermore, the query formulation from the short description and keyword may not be representative enough to recreate real-world scenario. Further experiments could use real query logs to improve the approach.

VIII. Related Work

For a large and evolving software system, the project team could receive bug reports over a long period of time. Many studies have been carried out on bug reports. For example, assigning bug reports to the right developers based on their categorization [1, 25] and predicting the time used to fix the bugs [26, 27]. Podgurski et al. [28] propose an approach to classify reported software failures in order to prioritize them and diagnose their causes. Hooimeijer and Weimer [29] develop a model to predict bug report quality.

Moreover, there have been a number of previous studies on detecting duplicate bug reports. Runeson et al. [30] conduct one of the earliest researches on finding duplicate bug reports. Their method characterizes each bug report as a bag of words tokens and each word constitutes an element of the feature vector. Their study shows the cosine measurement is the best and is able to detect 40% of the supplicate bug reports. Wang et al. [31] propose another feature vector construction approach where each feature is computed by considering both the TF and IDF value of the words in the bug reports. Their approach could detect 67%–93% of the duplicate bug reports. More recently, Sun et al. [6] propose to use a Support Vector Machine (SVM) to train a model that would compute the probability of two reports being duplicated. However, all these approaches focus on identifying whether two bug reports are duplicates. Less attention has been paid to examine whether the same bug has been reported before a new bug report is submitted.

Latent Semantic Analysis (LSA) or LSI has been extensively researched since 1990s [32, 33]. LSI uses SVD to decompose the term-document matrix so as to expose the hidden semantic structure. The decomposed matrices can be used to construct a low-dimension approximation of the original matrix. LSI has been proved to be effective in text classification [34] and information retrieval [35]. Unlike research conducted in [36], which concludes that LSI was outperformed by Log-Entropy based weighting scheme, in this paper we incorporate LSI as a feature into our LTR model.

As a machine learning technique, LTR aims at resolving an optimal model for a ranking problem. It emerged from late nineties and is becoming an intense research area for applications on a variety of machine learning problems. LTR has mostly been used in information retrieval to address the ranking of search results [13] and has been applied to real search engine applications. In our earlier work [10], we have presented a preliminary approach that applies LTR to the bug report search problem. We have proposed some specific features that can be extracted from the bug reports and queries. Preliminary experiments have been conducted to show the effectiveness of the proposed approach. Compared to our earlier work, the current approach is much more refined and enhanced with more thorough evaluation on larger datasets. In this work we have proposed more features (83 compared to 39) including the sums and standard variations of TF and IDF into the ranking model. In addition, we have incorporated the semantic features obtained from the analysis of Wikipedia and LSI to further improve our search model. In this paper, comprehensive experiments have been conducted by training the search model with larger training and testing datasets. We have also conducted experiments by using different feature categories and compared their effectiveness. Finally, we have inspected the weights of the features in the ranking models. The approach proposed in this paper achieves better performances than the previous work.

Recently, the ranking methods such as the extended BM25F [7] and LTR [8] are also used for duplicate bug report detection. However, these work focus on duplicate bug report detection, which can be viewed as a way of remediation. None of them aims at improving the search quality in bug tracking systems with the aim of ranking the previously reported bugs higher so that duplicate reports can be prevented from submission. Additionally, all the existing duplicate bug detection methods mentioned above make use of the whole report. Different from these methods, our method retrieves previously reported bug reports based on short user queries. Gottipati et al. [37] propose using manually tagged training set of software forum posts to train a classifier for identifying answers to each question. Our approach differs from theirs in that we aim to improve the search quality of bug report system instead of QA forums, by making innovative use of the relatively effective LTR ranking method as opposed to classification.

IX. Conclusions

The goal of this paper is to improve the search quality of a bug tracking system. We propose an approach based on a more effective ranking method Ranking SVM. We identify 83 textual, categorical and semantic features. Based on these features, a Ranking SVM based learning to rank model is trained. Given a short user query, our work retrieves previously reported bug reports. Our experiments on Eclipse and Mozilla projects show that the proposed approach can enhance the search quality of the existing bug tracking system by retrieving more relevant bug reports. For example, for Eclipse and Mozilla, the MAP values are above 0.61. The average F-measure values for the top 10 returned results for Eclipse and Mozilla are above 0.80 and above 0.65, respectively.

The proposed approach can help a reporter locate potential relevant bug reports more precisely. Based on the search results, the reporter can determine if she/he would like to continue to report the bug. The approach has the potential to prevent the same bugs from being reported repeatedly, thus relieving the burden of developers from dealing with duplicate bug reports, and ultimately improving the productivity of software maintenance.

Currently, we have considered the features listed in Table II. In future we will continue exploring the features of bug reports to further improve the search quality of bug tracking systems. In addition, we will conduct more experiments on large-scale projects, including industrial applications.

REFERENCES

[1] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 361–370.

[2] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful? really?" in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 337–345.

[3] Enter A Bug. [Online]. Available: https://bugzilla.mozilla.org/-enter_bug.cgi

[4] Bug Writing Guidelines. [Online]. Available: https://bugs.eclipse.org/-bugs/page.cgi?id=bug-writing.html

[5] Memory leak on page after use of ShareThis. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=751656

[6] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 45–54.

[7] C. Sun, D. Lo, S. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 253–262.

[8] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st ACM International Conference on Information and knowledge management*. ACM, 2012, pp. 852–861.

[9] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 69–78.

[10] K. Liu, H. Tan, and M. Chandramohan, "Has this bug been reported?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 28.

[11] Okapi BM25. [Online]. Available: http://en.wikipedia.org/wiki/-Okapi_BM25

[12] Apache Lucene. [Online]. Available: http://lucene.apache.org/

[13] T. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[14] H. Li, "A Short Introduction to Learning to Rank," *IEICE Transactions on Information and Systems*, vol. 94, no. 10, p. 1854, 2011.

[15] Z. Cao, T. Qin, T. Liu, M. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *Proceedings of the 24th International Conference on Machine Learning*. ACM, 2007, pp. 129–136.

[16] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 133–142.

[17] Support Vector Machine for Ranking. [Online]. Available: http://-www.cs.cornell.edu/people/tj/svm_light/svm_rank.html

[18] Princeton University "About WordNet.". [Online]. Available: http://-wordnet.princeton.edu

[19] I. H. Witten and D. Milne, "An effective, low-cost measure of semantic relatedness obtained from wikipedia links," in *Proceeding of AAAI Workshop on Wikipedia and Artificial Intelligence: an Evolving Synergy, AAAI Press, Chicago, USA*, 2008, pp. 25–30.

[20] D. Milne and I. H. Witten, "Learning to link with wikipedia," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management*. ACM, 2008, pp. 509–518.

[21] WikipediaMiner. [Online]. Available: http://wikipedia-miner.cms.waikato.ac.nz/

[22] Snowball Stemmer. [Online]. Available: http://snowball.tartarus.org/

[23] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.

[24] A Stop Word List. [Online]. Available: http://jmlr.csail.mit.edu/papers/-volume5/lewis04a/a11-smart-stop-list/english.stop

[25] D. Cubranic, "Automatic bug triage using text categorization," in *Ín SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[26] G. Bougie, C. Treude, D. German, and M. Storey, "A comparative exploration of freebsd bug lifetimes," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*. IEEE, 2010, pp. 106–109.

[27] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 495–504.

[28] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2003, pp. 465–475.

[29] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.

[30] 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*. Ieee, 2007, pp. 499–510.

[31] 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 30th International Conference on Software Engineering*. ACM, 2008, pp. 461–470.

[32] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[33] T. Landauer, P. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[34] S. Zelikovitz and H. Hirsh, "Using lsi for text classification in the presence of background text," in *Proceedings of the 10th International Conference on Information and Knowledge Management*. ACM, 2001, pp. 113–118.

[35] S. Dumais *et al.*, "Latent semantic indexing (lsi) and trec-2," *NIST SPECIAL PUBLICATION SP*, pp. 105–105, 1994.

[36] N. Kaushik and L. Tahvildari, "A comparative study of the performance of ir models on duplicate bug detection," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 159–168.

[37] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 323–332.