Learning to Rank Duplicate Bug Reports

Jian Zhou and Hongyu Zhang
School of Software, Tsinghua University
Beijing 100084, China
zhoujian1286@yahoo.com.cn, hongyu@tsinghua.edu.cn

ABSTRACT
For a large and complex software system, the project team could receive a large number of bug reports. Some bug reports could be duplicates as they essentially report the same problem. It is often tedious and costly to manually check if a newly reported bug is a duplicate of an already reported bug. In this paper, we propose *BugSim*, a method that can automatically retrieve duplicate bug reports given a new bug report. BugSim is based on learning to rank concepts. We identify textual and statistical features of bug reports and propose a similarity function for bug reports based on the features. We then construct a training set by assembling pairs of duplicate and non-duplicate bug reports. We train the weights of features by applying the stochastic gradient descent algorithm over the training set. For a new bug report, we retrieve candidate duplicate reports using the trained model. We evaluate BugSim using more than 45,100 real bug reports of twelve Eclipse projects. The evaluation results show that the proposed method is effective. On average, the recall rate for the top 10 retrieved reports is 76.11%. Furthermore, BugSim outperforms the previous state-of-art methods that are implemented using SVM and $BM25F_{ext}$.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering;* H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing.

Keywords
Bug reports, duplicate bug retrieval, duplicate documents, software maintenance, learning to rank.

1. INTRODUCTION
With the rapid development of information technology, software systems are getting more and more complex. Although a range of measures have been taken to assure software quality, in reality released software systems still contain bugs. For a large-scale, widely-used software system, the project team could receive a large number of bug reports over time. For example, 49,422 bugs were reported for the Eclipse projects in 2010, in average 135 bugs every day. Software bugs are typically maintained by a bug tracking system such as BugZilla[1], which stores the reported bug reports and tracks their status. After a bug report is created by the bug tracking system, the project team could retrieve it, examine it, verify it, and assign it to a developer to work on.

The bug reporting process is essential an uncontrolled, distributed process. Users and testers at different sites can report bugs whenever they encounter them. Therefore, often some bug reports are duplicates because they actually describe the same problem. For example, among all Eclipse bugs reported in 2010, 3799 of them are duplicates. Studies also show that as many as 36% of bug reports could be duplicates or invalid [1]. The existence of duplicate bug reports can exacerbate the already high cost of software maintenance, and can even lead to wasted maintenance efforts (imaging a developer spends days on verifying and debugging a bug, and finally realizes that the same problem has been reported before by a different user and fixed by another developer).

To reduce software maintenance effort, users/developers are required to check whether a new bug report is a duplicate of an existing one or not before reporting/handling a new bug. As the number of bug reports maintained by a bug tracking system could be very large, a manual process could be tedious and error-prone. In recent years, researchers have proposed automated methods for detecting duplicate bug reports. Given a new bug report, these methods analyze the existing bug reports stored in a bug track system, and return the top *N* related bug reports using certain information retrieval techniques. For example, [20] uses a SVM-based discriminative model to compute the probability of two bug reports being duplicate and then retrieval the top *N* most similar bug reports as the candidate duplicate bug reports. [21] uses an extended BM25F model to retrieve related bug reports.

In this paper, we propose *BugSim*, a duplicate bug retrieval method based on the concept of *learning to rank*. We first collect a training set, which consists of triples (*bug, dup bug, non-dup bug*), where *dup bug* represents a duplicate of the *bug, non-dup bug* represents a non-duplicate of the *bug.* We then create a ranking model to rank the *dup bug* higher than the *non-dup bug.* To construct the ranking model, we identify 9 textual and statistical features of bug reports, and learn the weights of the features by applying the stochastic gradient descent algorithm over the training dataset. For a new bug report, we retrieve and rank the duplicate bug reports using the trained model.

We evaluate our method using more than 45,100 real bug reports of twelve Eclipse projects. The evaluation results show that the proposed method is effective. The recall rates for the top 10 retrieved reports are above 61%, with an average of 76.11%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'12, October 29–November 2, 2012, Maui, HI, USA.
Copyright 2012 ACM 978-1-4503-1156-4/12/10...$15.00.

[1] http://www.bugzilla.org/

Furthermore, BugSim outperforms the previous SVM-based method by 15.41% and the BM25F$_{ext}$-based method by 3.71% on the recall rate of the top 10 results.

The organization of the paper is as follows. In Section 2, we describe the background of this work. In Section 3, we describe the proposed BugSim approach. Section 4 describes our experimental design, and Section 5 shows and discusses the experimental results. Section 6 gives the threats to validity. We discuss the related work in Section 7 and conclude the paper in Section 8.

2. BACKGROUND

2.1 Duplicate Bug Reports

During software testing and maintenance, a large number of bugs could be discovered and reported. Different users/testers may report the same problem at different time, thus causing duplicate bug reports. Figure 1 gives an example of duplicate bug reports. Bug #214445[2] and Bug #214451[3] are real bug reports for the Eclipse project. Each bug report contains a *summary* and a *description*, and is written in natural language. We can see that the two bug reports contain many similar words such as *select*, *value*, *filter*, *table*, etc. Usually, duplicate bug reports exhibit high degree of text similarity as they report the same problem. Therefore, information retrieval (IR) techniques can be applied to help developers detect duplicate bug reports.

Bug #214445, reported by ysun, on 2008-01-06

Summary: <Select value...>cannot select any value.

Description: <Select value...>cannot select any value [00].

Step:1. New a report. 2. Add a data source and dataset. 3. Add a table. 4. Add a filter/map/highlight. 5. Select a column and open the select value dialog. 6. Select one value and click OK.

Actual result: Nothing value is selected.

Bug #214451 , reported by Wen Lin, on 2008-01-07

Summary: Select value in table filter condition panel does not take any effect.

Description: Select value in table filter condition panel does not take any effect...

Step to reproduce: 1. New a table binding to a dataset in a report. 2. Add a filter in filters panel. 3. Set a filter condition as row "xxx" not equal to a value; select this value from select value panel.

Expect result: The value selected by user should be set to the condition combo.

Actual result: The select value operation does not take any effect.

Figure 1. An example of duplicate bug reports

2.2 Duplicate Bug Report Detection

Recently, researchers have proposed some IR-based methods for automated retrieval of duplicate bug reports. A general process of these methods is shown in Figure 2. The existing bug reports are stored at a repository managed by a bug tracking system. These bug reports are first preprocessed (including tokenization, stemming, and stop words removal). The features of the bug report are extracted and used to construct a retrieval model. When a new bug report arrives, the retrieval model compares its features to the features of the existing reports stored in the repository. The existing reports are then ranked according to their similarities to the new bug report, and are returned to the users. The users can examine the top *N* returned results and mark if the new bug report is a duplication.

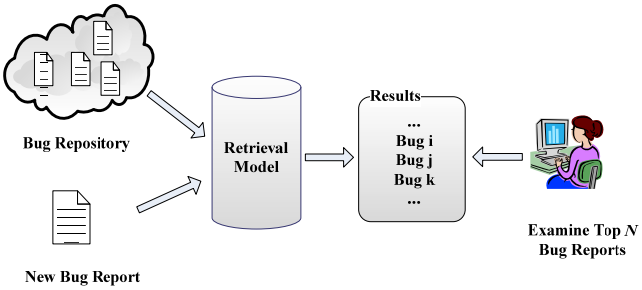

Figure 2. A general process for duplicate bug detection

The differences among the existing duplicate bug report detection methods are mainly in the types of features being selected and in the information retrieval techniques being applied. In this section, we briefly introduce two major state-of-art methods:

SVM:

In [20], the authors proposed a SVM-based discriminative model to determine how likely two bug reports are duplicates. They organize the existing bug reports as a list of buckets, where each bucket contains a set of confirmed duplicate bug reports. Pairs of duplicate and non-duplicate bug reports can be extracted from the buckets and used for training the model. As a bug report consists of two textual fields: summary and description, they identify three bags of words: summary, description, and both (summary + description). For each bag, an *idf* measure is computed, so there are three types of *idf* (*idf*-summary, *idf*-description, and *idf*-both). The total number of *idf* combinations is 27 for each pair of bug. Each *idf* combination is considered as a feature. They also consider bigrams besides unigrams, therefore making the total number of features 54. After identifying the features, they then build a SVM-based discriminative model, which can produce a probability of two bug reports being duplicates of each other. When a new bug report comes, the trained model is applied to retrieve a ranked list of candidate reports of which the new bug is likely a duplicate. Once developers confirm a duplication, the new bug report is placed into the corresponding bucket. Otherwise, a new bucket is created for the new bug.

BM25F$_{ext}$:

In [21], the authors proposed to use BM25F$_{ext}$, an extended BM25F model [17], to measure the textual similarity between two bug reports. They also organize the existing bug reports as a list of buckets and pairs of duplicate and non-duplicate bug reports are used for model training. They consider bug reports as structural documents that are composed of two fields (summary and description) with different degrees of importance, thus applying a BM25F model for structured document retrieval. Also, they consider using the entire bug report as a query, therefore they extend the BM25F model to incorporate the effect of long query. For a document *d* and a query *q*, the score of their extended model BM25F$_{ext}$ is computed as follows:

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=214445
[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=214451

$$BM25F_{ext}(d,q) = \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k_1 + TF_D(d,t)} \times W_Q$$

$$\text{, where } W_Q = \frac{(k_3+1) \times TF_Q(q,t)}{k_3 + TF_Q(q,t)}$$

$$TF_Q(q,t) = \sum_{f=1}^{k} w_f \times occurrences(q[f],t) \quad (1)$$

$$TF_D(d,t) = \sum_{f=1}^{k} \frac{w_f \times occurrences(d[f],t)}{1 - b_f + b_f \times \frac{length_f}{average\_length_f}}$$

, where f represents a field (either bug summary or bug description). For each field f, $length_f$ is the length of the field, $average\_length_f$ is the average length of f across all documents in the corpus, w_f is the field weight, and b_f is a scaling parameter ($0 \le b_f \le 1$). *IDF* is the inverse document frequency and *TF* is the term frequency. k_1 and k_3 are tuning parameters that control the effect of term frequency. Unlike the classic BM25F function, $BM25F_{ext}$ considers the weight of terms in queries by introducing an additional parameter k_3.

In [21], the authors also propose a retrieval function called REP. In REP, they identify seven features of bug reports, including textual similarity computed using $BM25F_{ext}$ and several categorical related features. The weight of each feature is tuned using the gradient descent algorithm. Finally, the reports in all buckets are ranked by the weighted sum of feature scores and are returned to developers. However, many of the categorical features (such as bug priority, buggy component, type, and version) may be unknown at the time of reporting bug, especially to end users. Therefore, we only consider the $BM25F_{ext}$ model in this paper.

2.3 Learning to Rank

As a relative new form of statistical learning, learning to rank techniques have received much attention in recent years. Learning to rank is a supervised (or semi-supervised) learning method, which learns a model from ordered objects in a training set and uses the model to rank unknown objects.

The information retrieval (IR) problem can be transformed as the problem of ranking. All documents in the corpus are ranked according to their relevancy to the query, highly relevant documents should be ranked higher than the irrelevant ones. Learning to ranking algorithms such as RankBoost [8], RankSVM [10], and RankNet [4] construct pairs between documents and use machine learning technique to minimize the number of disordered pairs. These algorithms are often called pairwise approaches and have been applied to retrieve documents [4, 5, 13]. More recently, researchers also proposed Listwise approaches [6, 28, 29], which aim to directly minimize the loss function defined on a list of objects.

In our approach, we treat existing bug reports stored in a bug tracking system as documents and a newly arrived bug report as a query. We design methods to retrieve duplicate bug reports by using learning to rank techniques. We apply the pairwise approach, as it is difficult to obtain the complete order of duplicate bug reports, but it is easier to know the partial orders between two bug reports.

3. THE PROPOSED METHOD

3.1 Overall Structure

In this section, we present a method called *BugSim* for ranking duplicate bug reports for a software project. Figure 3 shows an overall structure of BugSim. Following [20], we organize the existing bug reports as a list of buckets, where each bucket contains a set of confirmed duplicate bug reports. A bug report cannot belong to any two buckets thus each bucket is distinct. If a bug does not have duplicates, it corresponds to a bucket that only contains itself.

From the buckets, we collect a training set consisting of triples (*bug, dup bug, non-dup bug*), where *dup bug* represents a duplicate of the *bug, non-dup bug* represents a non-duplicate of the *bug*. We first identify features of bug reports and propose a similarity function to measure the similarities between two bug reports based on these features. Based on the training set, we then construct a retrieval model, which ranks the *dup bug* higher than the *non-dup bug* and minimizes the loss function.

Given a new bug report, the retrieval model can rank the existing bug reports according to the similarity function, and present the ranked results to users as candidate duplicate bug reports. If the new bug report does not belong to any existing bucket, a new bucket is created for this bug. Otherwise, the bug is assigned to the bucket where the duplicate bug report is.

In the following subsections, we describe the feature selection, the model training, and the ranking processes in more details.

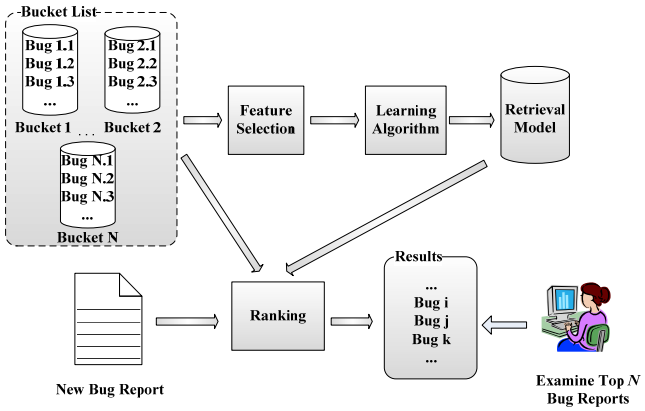

Figure 3. An overall structure of BugSim

3.2 Feature Selection

As described in Section 2, a bug report consists of two parts: summary and description. Inspired by the work on discriminative models for information retrieval [16], we identify 9 features based on bug summary and bug description as shown in Table 1. These features use statistics such as the word count and *idf* (inverse document frequency) and their combinations. We use these features to discriminate the bug reports.

In Table 1, S_1 and D_1 represent the summary and descriptions of a bug report B_1, respectively. S_2 and D_2 represent the summary and descriptions of a bug report B_2, respectively. $idf_s(w)$ represents the *idf* (Inverse Document Frequency) value of a word w over the summaries of all existing bug reports. $idf_D(w)$ represents the *idf* value of a word w over the descriptions of all existing bug reports. $c(w,D_2)$ represents the number of times the word w appears in D_2. $|D_2|$ means the length of D_2. $|C_D|$ means the total number of

unique words that appear in the descriptions of all existing bug reports. $c(w,C_D)$ means the number of times the word w appears in the descriptions of all existing bug reports.

Table 1. The identified features between two bug reports

#	Feature	#	Feature
1	$\sum_{w \in S_1 \cap S_2} idf_S(w)$	2	$\frac{\lvert S_1 \cap S_2 \rvert}{\lvert S_1 \cup S_2 \rvert}$
3	$\sum_{w \in S_1 \cap D_2} \log(c(w,D_2)+1)$	4	$\sum_{w \in S_1 \cap D_2} \log(\frac{c(w,D_2)}{\lvert D_2 \rvert}+1)$
5	$\sum_{w \in S_1 \cap D_2} \log(\frac{c(w,D_2)}{\lvert D_2 \rvert} \times idf_D(w)+1)$	6	$\sum_{w \in S_1 \cap D_2} idf_D(w)$
7	$\sum_{w \in S_1 \cap D_2} \log(\frac{c(w,D_2)}{\lvert D_2 \rvert} \times \frac{\lvert C_D \rvert}{c(w,C_D)}+1)$	8	$\sum_{w \in S_1 \cap D_2} \log(\frac{\lvert C_D \rvert}{c(w,C_D)}+1)$
9	$VSMScore(D_1, D_2)$		

Feature 1 describes the *idf* values of the words that are common to the summary of the two bug reports. Feature 2 describes the similarity between the summary of the two bug reports (measured in terms of the number of common words). Feature 3 describes the number of times words used in the summary of B_1 appearing in the descriptions of B_2. Feature 4 describes the frequency of words used in the summary of B_1 appearing in the descriptions of B_2. Feature 5 gives the multiplication of the term frequency and the *idf* values (*tf* * *idf*). Feature 6 gives the *idf* values of words that are used in both the summary of B_1 and the descriptions of B_2. Feature 7 describes the weights of words that are used in both the summary of B_1 and the descriptions of B_2, in the entire document collection. Feature 8 describes the inverse collection frequency of words that are used in both the summary of B_1 and the descriptions of B_2. Feature 9 gives the text similarity of the two bug reports, measured using VSM (Vector Space Model). For some features, the *log* function is used to dampen the effects of large numbers.

Based on the identified features, we define a metric *BugSim* for measuring the similarity between two bug reports B_1 and B_2 as follows:

$$BugSim(B_1, B_2) = \sum_{i=1}^{\#\ feature} w_i f_i^{B_1, B_2} \tag{2}$$

In (2), f_i represents the ith feature; w_i represents the weight of the ith feature. Higher *BugSim* values indicate higher similarity between the two bug reports.

3.3 Learning the Model

To apply (2), 9 weights should be given. It is difficult to obtain the values of these weights subjectively. In our approach, we adopt a learning process to tune these weights.

In our training set, each instance is a triple (*bug, dup bug, non-dup bug*), where *dup bug* represents a duplicate of the *bug, non-dup bug* represents a non-duplicate of the *bug.* Clearly, if we want the *dup bug* to be ranked higher than the *non-dup bug,* the followings should hold:

$$BugSim(bug, dup\ bug) > BugSim(bug, non\text{-}dup\ bug) \tag{3}$$

We also utilize the Fidelity loss function (F_{ij}) [25] to measure the cost of the *BugSim* on an instance I:

$$F_{ij} = 1 - \left(\left[P_{ij}^* \times (\frac{e^{o_{ij}}}{1+e^{o_{ij}}}) \right]^{\frac{1}{2}} + \left[(1-P_{ij}^*) \times (\frac{1}{1+e^{o_{ij}}}) \right]^{\frac{1}{2}} \right) \tag{4}$$
$$where\ o_{ij} = BugSim(bug, non\text{-}dup\ bug) - BugSim(bug, dup\ bug)$$

In (4), P^*_{ij} is the target probabilities that sample i is to be ranked higher than sample j. In the case of duplicate bug detection, duplicate bug reports should be always ranked higher than non-duplicate ones. So equation (4) can be simplified as follows:

$$F_{ij} = 1 - (\frac{1}{1+e^{o_{ij}}})^{\frac{1}{2}} \tag{5}$$

The Fidelity loss function was originally a distance metric in physics and was recently used by [25] in the probabilistic ranking framework. It has been shown that the Fidelity loss function exhibits properties that are helpful for ranking. From (5) we can see that the value of F_{ij} increases when o_{ij} increases.

To better differentiate *dup bug* and *non-dup bug*, the values of w_i should be chosen in such a way that the Fidelity loss function is minimized. To obtain the optimal values of w_i, we apply the stochastic gradient descent algorithm [15]. The stochastic gradient descent algorithm iterates N time. In each iteration, for each instance in the training set, the algorithm adjusts each weight as follows:

$$w_i = w_i - \eta \times \frac{\partial F_{ij}}{\partial w_i}(I) \tag{6}$$

, where η is the tuning rate, which is usually a very small number such as 0.001. After N iterations, the w_i values that lead to the minimum F_{ij} value are the optimal weight values. In (6), the partial derivative of the Fidelity loss function with respect to w_i is derived as follows:

$$\frac{\partial F_{ij}}{\partial w_k} = \frac{\partial F_{ij}}{\partial o_{ij}} \times \frac{\partial o_{ij}}{\partial w_k} = \frac{\partial (1-(1+e^{o_{ij}})^{-1/2})}{\partial o_{ij}} \times \frac{\partial o_{ij}}{\partial w_k}$$
$$= \frac{1}{2}(1+e^{o_{ij}})^{-3/2} \times e^{o_{ij}} \times \frac{\partial o_{ij}}{\partial w_k} \tag{7}$$
$$\frac{\partial o_{ij}}{\partial w_k} = \frac{\partial (BugSim(bug, non\text{-}dup\ bug) - BugSim(bug, dup\ bug))}{\partial w_k}$$
$$= f_k^{bug, non-dup\ bug} - f_k^{bug, dup\ bug}$$

In summary, Figure 4 shows the algorithm we used to perform training. The algorithm includes two parts: the selection of training instances and the learning of weights.

3.4 Ranking for New Bug Reports

After the ranking model is trained, we can use it for duplicate bug report retrieval. When a new bug report arrives, BugSim computes the similarity between a new bug report and a bucket, which is the

maximum similarity value between the new bug report and all the existing bug reports in the bucket. To compute the similarity between the new report and an existing one, BugSim first extracts the features between them (as described in Section 3.2), and then calculates the similarity using equation (2). BugSim ranks all the buckets by their similarity values to the new report. For each bucket, the most similar bug report is presented to the users. The users can then check if the new bug is a duplicate of a bug in the top *N* list. If it is a duplicate, it is inserted into the bucket that contains the bug it duplicates to. If it is not a duplicate bug, a new bucket is created for the new bug report.

TRAINING (*Buckets, S*)

Buckets: The list of buckets that contain duplicate bug reports.

S: The size of the training set

// Training set selection

1. Initialize the training set *T*
2. Randomly select two bug reports from the *Buckets* that contain more than one bug reports, and label them as *bug* and *dup bug.*
3. Randomly select a bucket that is different from the bucket that contains the *bug,* and randomly select one bug, label it as *non-dup bug*
4. Create a triple *I(bug, dup bug, non-dup bug)*, and add it into *T*.
5. Repeat the steps 2-4 until the size of *T* reaches *S.*

// Learning weights

6. Applying the stochastic gradient descent algorithm on *T* to tune the weight *w* of each feature
7. Return the tuned weights *w*

Figure 4. The training algorithm

4. EVALUATION

4.1 Dataset

To evaluate the effectiveness of the proposed approach, we choose the Eclipse[4] projects as our subject. Eclipse is an open source community whose projects are focused on building extensible software development platforms and frameworks. Software tools produced by the Eclipse projects are very popular and have millions of users worldwide[5]. The bugs reported against Eclipse are stored and tracked by the BugZilla tool[6]. Like in other projects, users who would like to post an Eclipse bug report are encouraged to check whether it has been posted already[7].

In our evaluation, we investigate 12 relatively independent projects (categories) in Eclipse, as shown in Table 1. These projects contain 45,170 bugs reported from January 1, 2008 to December 31, 2008. Among the 45,170 bugs, 2,949 are duplicates. Following the previous work [21], we construct a small training set, which contains total 5,863 bugs and 200 duplicate bugs (from 2008-01-01 to 2008-02-22). The remaining data is used as the testing dataset.

We use Java to implement the research prototype. The experiments are carried out on a Windows 2008 Server R2 with Intel Xeon(R) CPU E5506@2.13GHz and 8G memory.

Table 2. The Eclipse dataset

Project	Training Set		Test Set	
	#Duplicate Bugs	#Bugs	#Duplicate Bugs	#Bugs
Eclipse IDE	200	5863	928	9872
Eclipse Foundation			63	1497
DataTools			29	699
Tools			365	5257
BIRT			203	2989
RT			472	4702
DSDP			52	1321
TPTP			83	905
WebTools			226	3641
STP			3	129
Modeling			99	3239
Technology			226	5056

4.2 Research Questions

RQ1: How effective is BugSim?

RQ1 evaluates the effectiveness of our approach described in Section 3. To answer RQ1, we evaluate BugSim on the projects described in Table 2. We use the training set to train a ranking model, following the algorithm described in Figure 4. For each bug report in the testing set, BugSim retrieves the candidate duplicate bug reports and returns a ranked list to users. The performance of retrieval is measured using the recall rate and the MRR metrics.

RQ2: Can BugSim outperform the related methods?

RQ2 compares the proposed method with the two major related methods that are described in Section 2.2:

- SVM [20]: We use the *libsvm* tool [8] as the SVM implementation. Following [20], we identify 54 features to build a discriminative model on the training set. Once the model is created, we use it to determine the probability of a bug being a duplicate of the other.
- $BM25F_{ext}$ [21]: We use the parameters given in [21] to build a $BM25F_{ext}$-based model. $BM25F_{ext}$ is an extension of BM25F as described in Section 2.2. We use it to calculate the textual similarity between two bug reports and rank the candidate duplicate bugs according to the similarity values.

RQ3: Does the Fidelity loss function outperform the related RNC loss function?

[4] http://www.eclipse.org/
[5] http://www.numberof.net/number-of-eclipse-users/
[6] https://bugs.eclipse.org/bugs/
[7] http://wiki.eclipse.org/FAQ_How_can_I_search_the_existing_list_of_bugs_in_Eclipse%3F
[8] Libsvm: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

In our approach, we use the Fidelity function as the loss function. Other loss functions could be also adopted in learning to rank approaches. In RankNet [4], a pairwise cross entropy loss function is used to penalize ordering error, which is defined as:

$$C_{ij} = -P_{ij}^{*} O_{ij} + \log(1 + e^{O_{ij}}) \quad (8)$$

The above loss function is also used in [21, 23] and is referred to as the RNC function. This RQ compares the effectiveness of the Fidelity function with that of the RNC function.

4.3 Evaluation Metrics

We evaluate the performance of BugSim using the following metrics:

- Recall@*N*: The percentage of bug reports whose relevant duplicates can be detected in the top *N* (*N* = 1, 5, 10, 20) of the returned results. A better duplicate bug retrieval method should allow users to identify more duplicate bug reports by examining less number of bug reports. Thus, the higher the metric value, the better the retrieval performance.
- MRR (Mean Reciprocal Rank), which is a statistic for evaluating a process that produces a list of possible responses to a query [26]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries *Q*:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (9)$$

The higher the MRR value, the better the retrieval performance. Note that MRR is equal to MAP (Mean Average Precision), when only the first relevant answer is considered.

5. EVALUATION RESULTS

5.1 Results of RQs

RQ1: How effective is BugSim?

Table 3 shows the performance of BugSim on 12 projects, when recall rate is concerned. For the STP project, BugSim can successfully rank all its three duplicate bug reports as top 1 (the first returned result). Therefore its Recall@1 is 1. For the Eclipse IDE project, it has 9872 bugs and 928 of them are duplicates. Using BugSim, 45.58% duplicate bug reports can be detected at top 1, 71.77% can be detected within top 10, and 76.51% can be detected within top 20. For all the projects, the Recall@1 values are above 33% (with an average of 49.48%), the Recall@10 values are above 61% (with an average of 76.11%), and the Recall@20 values are above 67% (with an average of 82.03%). These results are considered satisfactory.

Table 3 also shows the performance of BugSim measured in terms of MRR. For the STP and Modeling projects, the MRR values exceed 0.7. For the rest of the projects, the MRR values are above 0.42.

In general, the evaluation results show that BugSim is effective in retrieving duplicate bug reports. It can allow developers detect duplicate bug reports by examining a small number of bug reports. The effort spent on bug handling can be reduced.

Table 3. The performance of BugSim

Project	Recall @1	Recall @5	Recall @10	Recall @20	MRR
BIRT	38.92%	66.01%	70.94%	80.30%	0.5054
DataTools	41.38%	68.72%	82.76%	86.21%	0.5434
DSDP	40.38%	76.92%	78.85%	82.69%	0.5515
Eclipse Foundation	44.44%	68.25%	76.19%	84.13%	0.5576
Eclipse IDE	45.58%	64.55%	71.77%	76.51%	0.5447
Modeling	71.72%	85.86%	88.89%	91.92%	0.7636
RT	33.26%	51.91%	61.23%	67.58%	0.4274
STP	100%	100%	100%	100%	1
Technology	46.02%	66.37%	73.01%	80.97%	0.5574
Tools	52.05%	72.33%	78.90%	84.93%	0.6148
TPTP	36.14%	56.63%	62.65%	73.49%	0.4564
WebTools	43.81%	61.50%	68.14%	75.66%	0.5226
average	*49.48%*	*69.92%*	*76.11%*	*82.03%*	*0.5871*

RQ2: Can BugSim outperform the related methods?

Table 4 shows the comparison between BugSim and SVM and $BM25F_{ext}$ based methods, when MRR is concerned. BugSim performs better than the SVM-based discriminative model on all experimental projects. The MRR values SVM achieves are between 0.32 and 0.73, with an average of 0.45. The improvements of BugSim over SVM range from 4.87% to 76.47%, with an average of 31.49%.

Table 4. The comparison between BugSim and the related work (MRR)

Project	BugSim	SVM		$BM25F_{ext}$	
		MRR	%impr.	MRR	%impr.
BIRT	0.5054	0.3159	60.03%	0.4763	6.11%
DataTools	0.5434	0.4378	24.12%	0.5068	7.23%
DSDP	0.5515	0.4883	12.94%	0.5431	1.54%
Eclipse Foundation	0.5576	0.4678	19.21%	0.4897	13.86%
Eclipse IDE	0.5447	0.3987	36.62%	0.5211	4.53%
Modeling	0.7636	0.7282	4.87%	0.7772	-1.75%
RT	0.4274	0.3265	30.93%	0.3870	10.43%
STP	1.0000	0.5667	76.47%	0.8333	20.00%
Technology	0.5574	0.4002	39.29%	0.5146	8.32%
Tools	0.6148	0.4666	31.77%	0.5744	7.03%
TPTP	0.4564	0.3710	22.99%	0.4438	2.84%
WebTools	0.5226	0.3901	33.96%	0.5264	-0.72%
average	*0.5871*	*0.4465*	*31.49%*	*0.5495*	*6.84%*

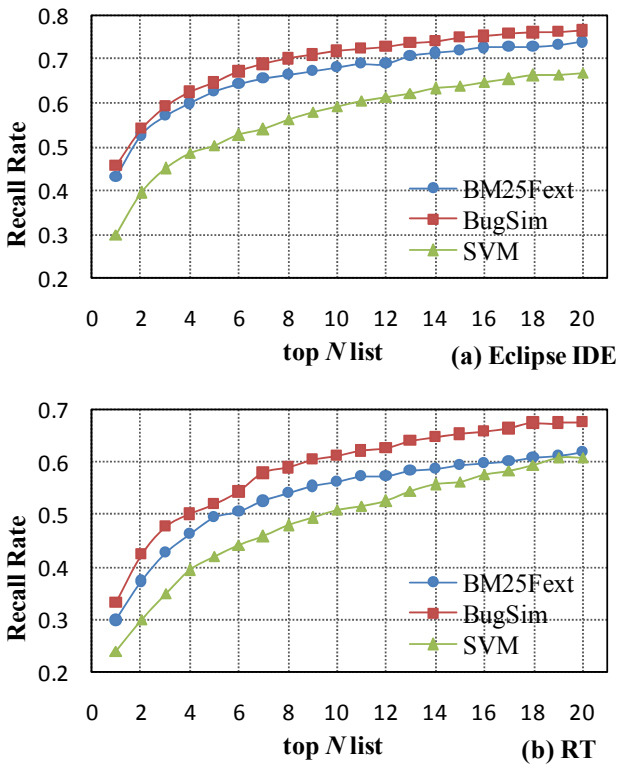

Figure 5. The comparisons between BugSim and the related work on Eclipse IDE and RT projects

BugSim also outperforms $BM25F_{ext}$ on 10 projects. The improvement is from 1.5% to 20%. BugSim only performs slightly worse than $BM25F_{ext}$ on two projects (1.7% and 0.7%, respectively). The average improvement of BugSim over $BM25F_{ext}$ is 6.84%. We also conduct a paired t-test and the result confirms that BugSim performs statistically significantly better than $BM25F_{ext}$, at the significance level 0.05 (with p value 0.017).

We also find that BugSim outperforms SVM and $BM25F_{ext}$, when performance is measured in terms of recall rate. For example, Figure 5 shows the results on Eclipse IDE and RT projects, at different top *N* lists. For the two projects, BugSim improves the SVM method by 14.15% – 53.26% and 9.75% – 29.00%, respectively, and improves $BM25F_{ext}$ by 2.59% – 5.47% and 4.49% – 12.00%, respectively. Table 5 gives the details on the percentage of improvement BugSim over SVM and $BM25F_{ext}$ on Recall@10 values, for all experimental projects. The average improvement of BugSim over SVM and $BM25F_{ext}$ is 15.41% and 3.71%, respectively.

RQ3: Does the Fidelity loss function outperform the related loss functions?

Figure 6 shows the impact of two different loss functions (Fidelity loss and RNC) on BugSim. Fidelity loss outperforms RNC when recall rate is concerned. For example, on the Eclipse IDE project, Fidelity loss function leads to 14.89% – 31.37% higher recall values than RNC. On the RT project, Fidelity loss function leads to 18.05% – 31.34% higher recall values than RNC.

Similarly, Fidelity loss function also outperforms RNC when the effectiveness is measured in terms of MRR (Figure 7). Fidelity loss leads to 4.07% – 27.78% higher MRR values on 10 projects.

On STP and DSDP projects, the two loss functions obtain similar performance. A paired t-test confirms that the results are statistically significant, at the significance level 0.05(with p value 0.0002).

Table 5. The comparison between BugSim and the related work (Recall@10)

Project	Bug-Sim	SVM		$BM25F_{ext}$	
		Recall@10	%impr.	Recall@10	%impr.
BIRT	70.94%	50.74%	39.81%	67.98%	4.35%
DataTools	82.76%	62.07%	33.33%	75.86%	9.10%
DSDP	78.85%	71.15%	10.82%	75%	5.13%
Eclipse Foundation	76.19%	73.02%	4.34%	73.02%	4.34%
Eclipse IDE	71.77%	59.38%	20.87%	68.10%	5.39%
Modeling	88.89%	80.81%	10.00%	86.87%	2.33%
RT	61.23%	50.85%	20.41%	56.36%	8.64%
STP	100%	100%	0.00%	100%	0.00%
Technology	73.01%	57.52%	26.93%	70.80%	3.12%
Tools	78.90%	69.32%	13.82%	76.16%	3.60%
TPTP	62.65%	59.04%	6.11%	61.45%	1.95%
WebTools	68.14%	57.52%	18.46%	69.03%	-1.29%
average	*76.11%*	*65.95%*	*15.41%*	*73.39%*	*3.71%*

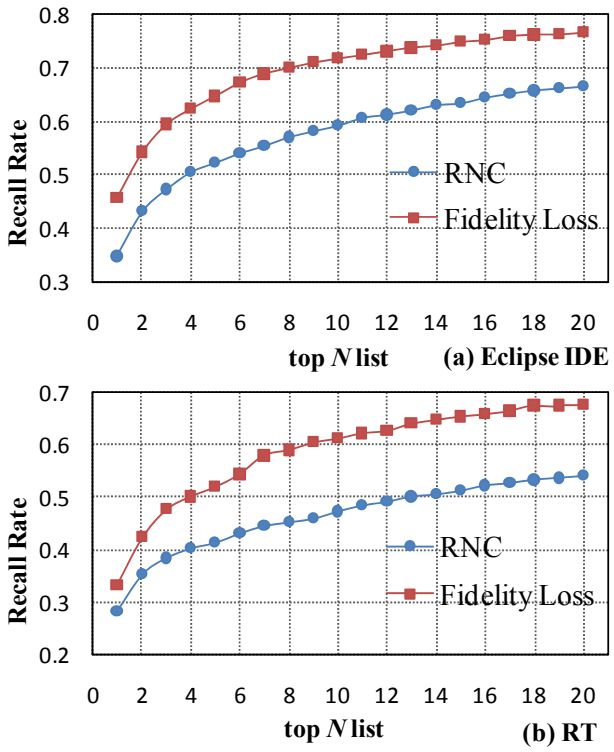

Figure 6. The impact of Fidelity Loss and RNC functions on BugSim (recall rate)

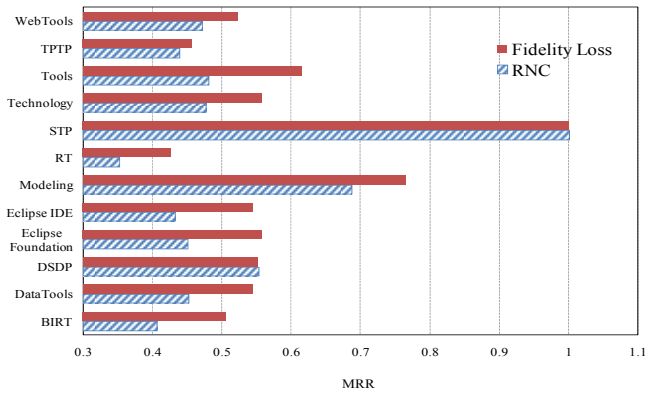

Figure 7. The impact of Fidelity Loss and RNC functions on BugSim (MRR)

5.2 Discussions

5.2.1 Why does BugSim perform better?

In Section 5.1, we have shown that BugSim outperforms SVM and $BM25F_{ext}$ based methods for detecting duplicate bug reports. We have identified the following major differences between BugSim and the two previous methods:

1) Feature selection: Inspired by the work on discriminative models for information retrieval [16], BugSim uses a much richer feature set that consists of both text similarity feature (feature 9 in Table 1) and statistical features (features 1-8 in Table 1). The statistical features consider both bug summary and bug description. Generally, a bug summary is short and a bug description is more detailed. Therefore, we treat a summary as a query and descriptions as documents, and identify features (features 3-8) to measure the similarity between a bug summary and a description. To measure the similarity of two bug summaries, we use two features (features 1-2) based on the *idf* value and the overlap of common words. To measure the similarity between two bug descriptions, we use a VSM model. In this way, we use different metrics to measure the similarity between different aspects of bug reports. We should note that in [21], the authors also identified several categorical features of bug reports, such as bug priority, buggy component, type, version, etc. The retrieval performance could be further improved by 2% (in terms of MRR) if these features are considered [21]. However, in real settings the values of these features could be unknown at the time of reporting bug, especially to end users. Therefore, in our work, we only consider the text similarity between two bug reports, which can be derived from natural language text only and is independent of bug tracking tools or processes.

2) Learning method: BugSim is essentially a pairwise learning to rank method. The previous SVM work [20] is based on classification of pairs of bug reports and can be considered as a form of pointwise learning to rank method. As pointed out by [14], the pointwise approach assumes relevance is absolute and query independent, which may bring bias to the training model. Also, BugSim utilizes the Fidelity loss function, while the previous $BM25F_{ext}$ work used the RNC loss function. As discussed in [25], Fidelity loss function possesses several new properties that are helpful for ranking (e.g., having a finite upper bound that is necessary for conducting query-level normalization) and therefore improves ranking performance.

5.2.2 The missing cases

Although our experiments show that BugSim can effectively retrieve duplicate bug reports, it may still fail to identify some duplicate bug reports. BugSim assumes that duplicate bug reports have high degree of text similarity, which is a valid assumption for most of duplicate bug reports. However, we found that some duplicate bug reports may use different words to describe the same problem. For example, when reporting the problem of a software crash, different reporters may use different words such as "exception", "crash", "failure", "error", "down", etc. Currently BugSim does not consider the semantic similarity among words. Therefore, it may treat these reports as dissimilar texts.

Another reason of missing cases is that some bug reports are very brief, as the users did not spend much time on writing detailed problem scenarios. This also causes difficulty in statistical analysis. We will address these two issues in our future work and to further improve the performance of the proposed approach.

5.2.3 Comparison to the Fingerprint-based methods

BugSim and the related methods (such as the SVM-based discriminative model and the $BM25F_{ext}$ model) basically detect near-duplicate documents by comparing the text similarity between pairs of bug reports using information retrieval techniques. Another major approach to the detection of similar documents is a technique known as *fingerprinting*, which represents and compares documents using compact values computed by certain hash functions. Examples of Fingerprint-based methods include Shingling [3, 9] and SimHash [7, 9, 19].

We compare the performance of BugSim to that of SimHash on the Eclipse projects. The results are shown in Figure 8. The MRR values of SimHash are ranging from 0.15 to 0.57 (with an average of 0.29), while the MRR values of BugSim are ranging from 0.43 to 1.00 (with an average of 0.59). A paired t-test also confirms the statistical significance ($p = 0.00$). Clearly, BugSim is more capable of detecting duplicate bug reports than SimHash. According to the literature, SimHash is effective in detecting duplicate web pages at a very large scale [9]. The duplicate web pages are identical/near-identical web pages stored at different locations. While in the cases of bug reports, although duplicate bug reports describe the same problem, they could differ in the detailed natural language descriptions.

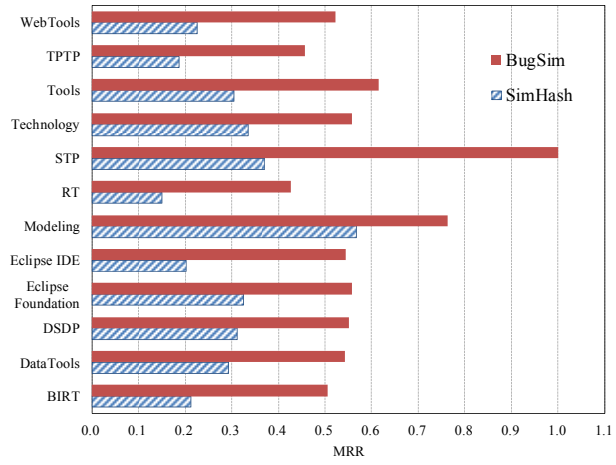

Figure 8. The comparison between BugSim and SimHash (MRR)

6. THREATS TO VALIDITY

There are potential threats to the validity of our work:

- The performance of our approach relies on bug report quality. A "bad" bug report could cause misclassification of the bug report. If a bug report does not provide enough information, or provides misleading information, the performance of BugSim is adversely affected.
- The dataset used in our experiments is collected from the Eclipse open source projects. We need to evaluate the performance of the proposed method on more types of projects, especially on commercial projects. This remains as an important future work.
- In our experiments, we evaluate BugSim on 45,100+ real bug reports of twelve Eclipse projects. We assume that the number of bug reports for a software system is limited. If the scale of a bug repository is very large (say consisting of millions of bug reports), the performance of BugSim could be affected. However, in real projects the number of bugs in a single software system is usually limited.

7. RELATED WORK

7.1 Duplicate document detection

Duplicate documents are often found in large corpus. Many researchers have proposed methods for detecting duplicate or near-duplicate documents. For example, Broder et al. [3] used word sequences (shingles) and a clustering algorithm to efficiently detect near-duplicate web pages. Brin et al. [2] proposed to use word sequences to detect copyright violations. Theobald et al. [24] proposed SpotSig, an algorithm for extracting and matching signatures for near duplicate detection in large Web crawls. Sood and Loguinov [19] used SimHash [7] to detect similar document pairs in large-scale collections.

The above methods are based on documents' fingerprints. Hoad and Zobel [11] also experimented with information retrieval based method for identifying versioned and plagiarized documents. They proposed several term-weight based similarity measures (called identity measures) to rank similar documents. Their experiments on two document collections showed that both the identity measures and the fingerprint-based methods can identify duplicate documents, and in general identity measures outperform fingerprint-based methods in terms of accuracy.

In this work, we focus on a special form of documents – bug reports. Unlike duplicate web pages or plagiarized documents, duplicate bug reports are not identical. The scale of bug reports for a software system is usually much smaller than the scale of a Web corpus. We identify statistical features of bug reports and apply learning to rank techniques to detect duplicate bug reports.

7.2 Duplicate bug detection

Recently, some researchers have proposed methods for detecting duplicate bug reports. We have already discussed and compared the SVM-based discriminative method and the $BM25F_{ext}$ method in previous sections. Runeson et al. [18] also treated bug reports as text documents and applied the Vector Space Model to retrieve duplicate bug reports. They computed a term's weight as *1+log(frequency)*, and applied three measures (cosine measure, dice coefficient, and Jaccard coefficient) to measure text similarity. Their experiments on Sony Ericsson mobile systems showed that about 40% of the duplicates could be found. Sun et al. [20] found that their SVM-based discriminative method outperforms Runeson et al.'s method.

Wang et al. [27] proposed to detect duplicate bug reports using both natural language text and execution information. However, their approach is limited in that not all bug reports come with runtime program execution information. Jalbert and Weimer [12] built a classifier to detect duplicate bug reports. They used bug reports' surface features, text similarity metrics, and graph clustering algorithms to identify duplicate documents. Their evaluation on 29,000 bug reports from the Mozilla project showed that 8% of duplicate reports can be successfully detected.

Recently, Sureka and Jalote [22] proposed to use n-grams based features to characterize a bug report. Their experiment on Eclipse projects showed that their approach is able to detect duplicate bug reports with reasonable accuracy. They randomly selected 1100 duplicate Eclipse bug reports and found that their approach can achieve Recall@10 value 21% and Recall@20 value 25%. Sun et al. [21] reported that their $BM25F_{ext}$ method outperforms Sureka and Jalote's method.

As described in previous sections, BugSim outperforms the existing state-of-art SVM and $BM25F_{ext}$ based methods (therefore other related methods described above). On Eclipse projects, the Recall@10 values BugSim achieves are between 61% and 100%.

8. CONCLUSIONS AND FUTURE WORK

Although quality control measures have been taken, in reality, software systems contain bugs. For a large-scale, widely-used software system, the project team could receive a large number of bug reports over time. The existence of duplicate bug reports could greatly increase software maintenance efforts. In this paper, we have proposed *BugSim*, a duplicate bug retrieval method based on learning to rank techniques. BugSim constructs a training set by assembling pairs of duplicate and non-duplicate bug reports. A training model is built using textual and statistical features of bug reports and a similarity function based on these features. Our experiments on 12 Eclipse projects show that BugSim is effective in retrieving duplicate bug reports. The recall rates for the top 10 retrieved reports are above 61%, with an average of 76.11%. Our method also outperforms the related state-of-art methods (SVM and $BM25F_{ext}$).

In the future, we will consider the semantic similarity between bug reports, aiming to further improve the performance of the proposed approach. We will evaluate the proposed approach on industrial projects. We also plan to integrate our approach into an existing bug tracking system.

ACKNOWLEDGEMENTS

This research is supported by the NSFC grant 61073006, and the Tsinghua University project 2010THZ0. We thank the authors of [21] for providing us the tool and the Eclipse dataset used in their experiments.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. 2006. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering* (ICSE '06). Pages: 361-370.

[2] S. Brin, J. Davis, and H. Garcia-Molina. 1995. Copy detection mechanisms for digital documents. In *Proc. ACM SIGMOD Annual Conference*. Pages: 398-409.

[3] A. Z. Broder, S.C. Glassman, M.S. Manasse, and G. Zweig. 1997. Syntactic Clustering of the Web. In *Proc. 6th International World Wide Web Conference.* Pages: 393-404.

[4] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning* (ICML '05). Pages: 89-96.

[5] Y. Cao, J. Xu, T-Y Liu, H. Li, Y. Huang, and H-W Hon. 2006. Adapting ranking SVM to document retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '06). Pages: 186-193.

[6] Z. Cao, T. Qin, T-Y Liu, M-F Tsai, and H Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning* (ICML'07). Pages: 129-136.

[7] M. S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing* (STOC '02). Pages: 380-388.

[8] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. 2003. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res. 4* (December 2003). Pages: 933-969.

[9] M. Henzinger. 2006. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '06). Pages: 284-291.

[10] R Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. *Advances in Large Margin Classifiers* (2000). Volume: 88, Issue: 2, Publisher: MIT Press, Pages: 115-132.

[11] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology* (2003). Volume 54, Issue 3, pages 203–215.

[12] N. Jalbert and W. Weimer. 2008. Automated Duplicate Detection for Bug Tracking Systems. In *Proceedings of the International Conference on Dependable Systems and Networks* (DSN '08). Pages: 52-61.

[13] T. Joachims. 2002. Optimizing search engines using click through data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (KDD '02). Pages: 133-142.

[14] T-Y Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval*. Vol. 3: No 3. Pages: 225–331.

[15] T. M. Mitchell. *Machine Learning*. New York: McGraw -Hill, 1997. Pages: 88-95.

[16] R. Nallapati. 2004. Discriminative models for information retrieval. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '04). Pages: 64-71.

[17] S. Robertson, H. Zaragoza, and M. Taylor. 2004. Simple BM25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management* (CIKM '04). Pages: 42-49.

[18] P. Runeson, M. Alexanderson, O. Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *Proceedings of the 29nd ACM/IEEE International Conference on Software Engineering* (ICSE '07). Pages: 499-510.

[19] S. Sood and D. Loguinov. 2011. Probabilistic near-duplicate detection using simhash. In *Proceedings of the 20th ACM international conference on Information and knowledge management* (CIKM '11). Pages: 1117-1126.

[20] C. Sun, D. Lo, X. Wang, J. Jiang, and S-C. Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE '10). Pages: 45-54.

[21] C. Sun, D. Lo, S-C. Khoo, and J. Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. *In Proc. Automated Software Engineering* (ASE'11). Pages: 253 – 262.

[22] A. Sureka and P. Jalote. 2010. Detecting duplicate bug report using character n-gram-based features. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference*. Pages: 366–374.

[23] M. Taylor, H. Zaragoza, N. Craswell, S. Robertson, and C. Burges. 2006. Optimisation methods for ranking functions with multiple parameters. In *Proceedings of the 15th ACM international conference on Information and knowledge management* (CIKM '06). Pages: 585-593.

[24] M. Theobald, J. Siddharth, and A. Paepcke. 2008. SpotSigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '08). Pages: 563-570.

[25] M-F. Tsai, T-Y. Liu, T. Qin, H-H. Chen, and W-Y. Ma. 2007. FRank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '07). Pages: 383-390.

[26] E.M. Voorhees. 1999. Question Answering Track Report. In *Proceedings of the 8th Text Retrieval Conference*. Pages: 77–82.

[27] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering* (ICSE '08). Pages: 461-470.

[28] F. Xia, T-Y. Liu, J. Wang, W. Zhang, and H. Li. 2008. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning* (ICML '08). Pages: 1192-1199.

[29] J. Xu and H. Li. 2007. AdaRank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '07). Pages: 391-398.