Towards More Accurate Retrieval of Duplicate Bug Reports

Chengnian Sun*, David Lo†, Siau-Cheng Khoo*, Jing Jiang†
*School of Computing, National University of Singapore
†School of Information Systems, Singapore Management University
suncn@comp.nus.edu.sg, davidlo@smu.edu.sg, khoosc@comp.nus.edu.sg, jingjiang@smu.edu.sg

***Abstract*—In a bug tracking system, different testers or users may submit multiple reports on the same bugs, referred to as duplicates, which may cost extra maintenance efforts in triaging and fixing bugs. In order to identify such duplicates accurately, in this paper we propose a retrieval function (REP) to measure the similarity between two bug reports. It fully utilizes the information available in a bug report including not only the similarity of textual content in *summary* and *description* fields, but also similarity of non-textual fields such as *product*, *component*, *version*, etc. For more accurate measurement of textual similarity, we extend *BM25F* – an effective similarity formula in information retrieval community, specially for duplicate report retrieval. Lastly we use a two-round stochastic gradient descent to automatically optimize REP for specific bug repositories in a supervised learning manner.**

We have validated our technique on three large software bug repositories from Mozilla, Eclipse and OpenOffice. The experiments show 10–27% relative improvement in *recall rate@k* and 17–23% relative improvement in *mean average precision* over our previous model. We also applied our technique to a very large dataset consisting of 209,058 reports from Eclipse, resulting in a *recall rate@k* of 37–71% and *mean average precision* of 47%.

I. Introduction

Due to complexities of software systems, software bugs are prevalent. To manage and keep track of bugs and their associated fixes, bug tracking system like Bugzilla[1] has been proposed and is widely adopted. With such a system, end users and testers could report bugs that they encounter. Developers could triage, track, and comment on the various bugs that are reported.

Bug reporting however is an uncoordinated distributed process. End users and testers might report the same defects many times in the bug reporting system. This causes an issue as different developers should not be assigned the same defect. Figuring out which bug reports are duplicate of others is typically done manually by a person called the *triager*. The triager would detect if a bug report is a duplicate; if it is, the triager would mark this report as a ***duplicate*** report and the first report as the ***master*** report. This process however is not scalable for systems with large user base as the process could take much time. Consider for example Mozilla; in 2005, it was reported that "everyday almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [1].

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

To address this issue there have been a number of studies that try to partially automate the triaging process. There are two general approaches: one is by filtering duplicate reports preventing them from reaching the triagers [2], the other is by providing a list of top-k related bug reports for each new bug report under investigation [3]–[6]. In this study, we focus on the second approach for the following reasons. The first approach is more difficult and the state-of-the-art approach proposed in [2] is only able to remove 8% of the duplicate reports. The remaining 92% of the duplicate bug reports would still require manual investigation. Furthermore, duplicate bug reports are not necessarily bad. Bettenburg et al. notice that one bug report might only provide a partial view of the defect, while multiple bug reports can complement one another [7]. Thus, in this study, we focus on providing a technique that could help in linking bug reports that are duplicate of one another.

Unfortunately, the accuracy of proposed techniques for duplicate bug report detection through ranking [3]–[6] is still low. In this work, we show an alternative approach with the goal of improving the accuracy of existing techniques. With a more accurate technique, triagers could be presented with better candidate duplicate bug report lists which would make his/her job easier.

Our approach is built upon $BM25F$ model which is studied not long ago in the information retrieval community [8], [9]. $BM25F$ is meant for facilitating the retrieval of documents relevant to a *short* query. In duplicate bug report detection, given a bug report (which is a rather *lengthy* query), we would like to retrieve related bug reports. We extend $BM25F$ model to better fit duplicate bug report detection problem by extending the original model to better fit longer queries.

Studies in [3], [5], [6] process only natural language text of the bug reports to produce a ranking of related bug reports. In this study we consider not only text but also other features that are available in BugZilla, $e.g.$, the component where the bug resides, the version of the product, the priority of the report. Wang et al. also include execution traces as information to predict duplicate bug reports [4]. We do not use execution traces in this study as they are hard to get and are often unavailable in typical bug reports.

We evaluate our approach on several large bug report datasets from large open source projects including Mozilla, a software platform hosting several sub-projects such as Firefox

browser and Thunderbird email client, Eclipse, a popular open source integrated development environment, and OpenOffice, a well-known open source rich text editor. For Eclipse we also consider a larger dataset that includes 209,058 bug reports. In terms of the types of programs and the number of bug reports considered for evaluation, to the best of our knowledge, our study is larger than any existing duplicate bug report detection studies in the literature. We show that our technique could result in 10–27% relative improvement in *recall rate@k* and 17-23% in *mean average precision* over state-of-the-art techniques [5], [6].

We summarize our contributions as follows:

1) We propose a new duplicate bug report retrieval model by extending $BM25F$. We engineer an extension of $BM25F$ to handle longer queries.
2) We make use of more information, that is easily available in bug tracking system (BugZilla), as compared to existing studies that rank bug reports. We use not only textual content of bug reports but also other categorial information including *priority*, *product version*, $etc.$
3) We are the first to analyze the applicability of duplicate bug report detection techniques on a total of more than 350,000 bug reports across bug repositories of various large open source programs including OpenOffice, Mozilla, and Eclipse.
4) We improve the accuracy of state-of-the-art automated duplicate bug detection techniques by 10-27% in *recall rate@k* ($1 \leq k \leq 20$) and 17-23% in *mean average precision*.

The paper is organized as follows. Section II presents some background information on duplicate bug reports, duplicate bug reports retrieval, BM25F and the optimization of ranking functions. Section III presents our approach to retrieve similar bug reports for duplicate bug report detection. Section IV describes our case study on more than 350,000 bug reports to show the utility of our proposed approach in improving the accuracy of existing approaches. Section V discusses related work, and finally, Section VI concludes and describes some potential future work.

II. Background

This section covers necessary background information. It first discusses duplicate bug reports, then describes the general workflow to retrieve duplicate bug reports. Next it introduces $BM25F$ – an effective textual similarity measure in information retrieval area, and lastly describes how to tune free parameters in a similarity function in order to achieve better performance.

A. Duplicate Bug Reports

A bug report serves multiple functions. It is used to file defects, propose features and record maintenance tasks. A bug report consists of multiple fields. The fields in different projects may vary to some extent, but in general they are similar. Table I lists the fields of interest to us in OpenOffice bug reports. Fields *summary* and *description* are in natural language text, and we refer to them as *textual features*, whereas the other fields try to characterize the report from other perspectives and we refer to them as *non-textual features* or *categorial features*.

TABLE I
Fields of Interest in an OpenOffice Bug Report

Field	Description
Summ	*Summary*: concise description of the issue
Desc	*Description*: detailed outline of the issue, such as what is the issue and how it happens
Prod	*Product*: which product the issue is about
Comp	*Component*: which component the issue is about
Vers	*Version*: the version of the product the issue is about
Prio	*Priority*: the priority of the report, $i.e.$, *P1*, *P2*, *P3*, $\cdots$
Type	*Type*: the type of the report, $i.e.$, *defect*, *task*, *feature*

In a software project, its bug tracking system is usually accessible to testers and even to all end users. Once a bug manifests, people can submit a bug report depicting the detail of the bug. However multiple reports from different submitters may correspond to the same bug, which causes the problem of duplicate bug reports. In this situation, the triager needs to label the duplicate reports as *duplicate* and add a link to the first report about the bug. We refer to the first report as ***master*** and the other duplicate ones as ***duplicate***.

Table II shows three pairs of duplicate reports in Issue Tracker of OpenOffice with the fields in Table I. The *description* of each report is not shown as it is long. As we can see, the two reports in each pair are similar in not only the *summary* field but also other fields. For example, the second pair has the same *component*, *priority* and *type*, and adjacent *version* in chronological order. This observation motivates us to consider these categorial features for duplicate bug report retrieval.

B. Workflow for Retrieving Duplicate Bug Reports

This section briefly describes the workflow to retrieve duplicate bug reports. More detail has been discussed in our previous paper [5].

In a duplicate report retrieval system, the fields *summary* and *description* of both existing and new bug reports are preprocessed by standard information retrieval techniques, $i.e.$, *tokenization*, *stemming* and *stop work removal*. Figure 1 depicts the overall flow. The bug repository is organized as a list of buckets – a hashmap-like data structure. The key of each bucket is a master report, and its value is a list of duplicate reports on the same bug. Each bucket has a distinct bug as its master report is not contained in other buckets. When a fresh bug report Q is submitted, the system computes the similarity between Q and each bucket, and returns K master reports, whose buckets have the top-K similarities. The similarity of a report and a bucket is the maximum similarity between the report and each report in the bucket computed by the component *Similarity Measure* in Figure 1.

In our previous work, we used a support vector machine (SVM) to train a model measuring the similarity between two reports [5]. The similarity is measured based on only

TABLE II
EXAMPLES OF DUPLICATE BUG REPORTS FROM OPENOFFICE ISSUETRACKER

Pair	ID	Summary	Product	Component	Version	Priority	Type
1	85064	[Notes2] No Scrolling of document content by use of the mouse wheel	Word	Code	680m240	P3	Feature
	85377	[CWS notes2] unable to scroll in a note with the mouse wheel	Word	UI	680m240	P3	Defect
2	85502	Alt+<letter> does not work in dialogs	None	UI	OOH680m4	P3	Defect
	85819	Alt-<key> no longer works as expected	Framework	UI	OOH680m5	P3	Defect
3	85487	connectivity: evoab2 needs to be changed to build against changed api	Database	None	OOH680m4	P1	Defect
	85496	connectivity fails to build (evoab2) in m4	Database	None	OOH680m4	P2	Defect

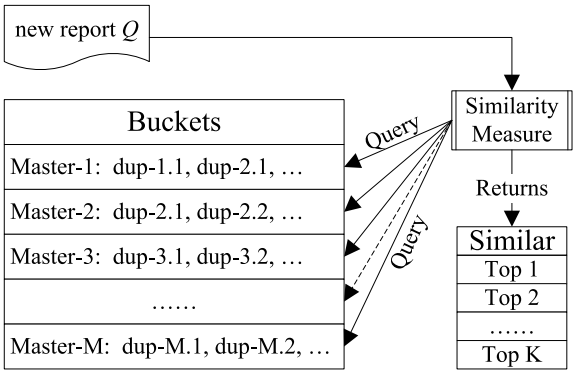


Fig. 1. Overall Workflow for Retrieving Duplicates

textual features: *summary* and *description*. While this approach has been shown to be effective, we believe it can be further improved. Thus in this paper we propose a new and comprehensive similarity function to substitute the component *Similarity Measure*. The other parts of the retrieval process remain unchanged.

C. BM25F

BM25F is an effective textual similarity function for structured document retrieval [8], [9]. A structured document is composed of several fields ($e.g.$, *summary* and *description* in a bug report) with possibly different degrees of importance. Given a document corpus D of N documents, each document d consists of K fields, and the bag of terms in the f-th field can be denoted by $d[f]$ where $1 \leq f \leq K$.

The definition of BM25F is based on two components. The first is the inverse document frequency (IDF) defined in (1), which is a *global* term-weighting scheme across documents,

$$IDF(t) = log\frac{N}{N_d} \tag{1}$$

where N_d is the number of documents containing the term t.

The other component is the *local* importance measure TF_D of a term t in a document d defined in (2), which is an aggregation of the local importance of t in each field of d.

$$TF_D(d,t) = \sum_{f=1}^{K} \frac{w_f \times occurrences(d[f],t)}{1 - b_f + \frac{b_f \times length_f}{average\_length_f}} \tag{2}$$

For each field f, w_f is its field weight; $occurrences(d[f],t)$ is the number of occurrences of term t in field f; $length_f$ is the size of the bag $d[f]$, $average\_length_f$ is the average size of the bag $d[f]$ across all documents in D, and b_f is a parameter $(0 \leq b_f \leq 1)$ that determines the scaling by field length: $b_f = 1$ corresponds to full length normalization, while $b_f = 0$ corresponds to term weight not being normalized by the length. (Length normalization of term weights is to mitigate the advantage that long documents have in retrieval over short documents.)

Based on the two components above, given a query q which can be a bag of words or a document, the BM25F score of a document d and q is computed as follows,

$$BM25F(d,q) = \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k_1 + TF_D(d,t)} \tag{3}$$

In (3), t is the shared term occurring in both d and q, and k_1 $(k_1 \geq 0)$ is a tuning parameter to control the effect of $TF_D(d,t)$. There is a set of free parameters to tune for BM25F to work most effectively for a certain document corpus, $i.e.$, w_f and b_f for each field f, and k_1. With K fields, there are $(1+2K)$ parameters in total. The following section introduces a technique available in information retrieval area to tune these parameters.

D. Optimizing Similarity Functions by Gradient Descent

This section briefly introduces an optimization technique in [10] based on stochastic gradient descent to tune parameters in a similarity function.

A function $sim(d,q)$ computes a similarity score between a document d and a query q, and has a vector of n free parameters $\langle x_1, x_2, \cdots, x_n \rangle$. In order for $sim()$ to perform most effectively for a document corpus, we need a training set to tune the n parameters in $sim()$. In the training set, each instance is a triple (q, rel, irr), where q is a query, rel is a document relevant to q, and irr is an irrelevant document. Ideally, $sim(d,q)$ should give a higher score to (rel,q) than to (irr,q), namely $sim(rel,q) > sim(irr,q)$.

In [10], Taylor et al. use a simplified version of RankNet cost function RNC presented in [11] to measure the cost of the application of $sim(d,q)$ on a training instance $I = (q, rel, irr)$.

$$RNC(I) = log(1+e^Y) \text{ where } Y = sim(irr,q) - sim(rel,q)$$

Lower RNC value means more accuracy of $sim()$, whereas higher value represents less accuracy of $sim()$. Intuitively, the RNC value for a training instance is large if the retrieval

function $sim()$ fails to rank the two documents rel, irr in the correct order, that is, $sim(irr, q) > sim(rel, q)$.

The whole optimization of $sim()$ is a process of minimizing the cost function RNC for each training instance, and the minimization is achieved by iteratively adjusting free parameters in $sim()$ via stochastic gradient descent.

Algorithm 1 Simplified Parameter Tuning Algorithm

TS: a training set
N: the times to iterate through TS
η: the tuning rate – a small number such as 0.001

```
for n = 1 to N do
    for each instance I ∈ TS in random order do
        for each free parameter x in sim() do
            x = x − η × ∂RNC/∂x (I)
        end for
    end for
end for
```

Algorithm 1 displays a simplified version of the tuning algorithm. Generally, the algorithm iterates through the training set N times, where N is provided by the user. At each time, for each training instance I, the algorithm adjusts each free parameter x to $x = x - \eta \times \frac{\partial RNC}{\partial x}(I)$. This adjustment is controlled by a small coefficient $\eta(\eta > 0)$ and $\frac{\partial RNC}{\partial x}$ – the partial derivative of RNC with respect to the free parameter x. In principle, the iterative adjustment of the value of x enables the latter to progress towards the minimum of RNC. For a textbook treatment on gradient descent, please refer to [12].

III. Approach

Our approach consists of three aspects. First we aim to improve the accuracy of textual similarity measures specially for bug reports. Thus we extend $BM25F$ – a successful measure in information retrieval area, by considering the weight of terms in queries. Second in order to enhance the performance of duplicate bug report retrieval, besides textual fields we try to take better advantage of the other kinds of information available in bug reports. Therefore we propose a new retrieval function which is a linear combination of similarities of textual and categorial features. Furthermore, to enable the retrieval function to work most effectively on a certain bug repository, we optimize it by performing gradient descent on a training set extracted from the bug repository. The following sections describe these aspects respectively.

A. Extending BM25F for Structured Long Queries

$BM25F$ is designed for short queries, which usually have no duplicate words. For example, the queries in search engines are usually of fewer than ten distinct words. However, in the context of duplicate bug report retrieval, each query is a new bug report. The query is structured as it is with a short summary and a long description, and it can sometimes be as long as more than one hundred words. We believe these two characteristics of bug report queries – structural and long – can further enhance the retrieval performance of $BM25F$, so we extend it into the following form by considering the term frequencies in queries.

$$BM25F_{ext}(d,q) = \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k_1 + TF_D(d,t)} \times W_Q$$
$$\text{where } W_Q = \frac{(k_3+1) \times TF_Q(q,t)}{k_3 + TF_Q(q,t)} \tag{4}$$

$$TF_Q(q,t) = \sum_{f=1}^{K} w_f \times occurrences(q[f], t) \tag{5}$$

In (4), for each shared term between a document d and a query q, its weight contains two components: one is the product of IDF value and term weight in d inherited from $BM25F$; and the other is the term weight in query q – W_Q. W_Q is derived from the query term weighting scheme of *unstructured* retrieval function Okapi BM25 [13], and it involves weight from the query computed by TF_Q defined in (5). The free parameter $k_3(k_3 \geq 0)$ is to control the contribution of the query term weighting, for example, if $k_3 = 0$, then the query term contributes no weight as W_Q becomes always equal to 1 and $BM25F_{ext}$ is reduced to $BM25F$. W_Q is monotonically increasing and concave with k_3, upper-bounded by TF_Q.

In (5), the weight of a term is the aggregation of the product of w_f – the weight of field f, and the number of occurrences of t in $q[f]$. Different from TF_D defined in (2), there is no length normalization of query terms as retrieval is being done with respect to a single fixed query.

Compared to $BM25F$, $BM25F_{ext}$ has an additional free parameter k_3, thus given documents with K fields, $BM25F_{ext}$ has $(2 + 2K)$ free parameters.

B. Retrieval Function

From the three pairs of duplicate reports shown in Table II, we note that duplicate reports are similar not only textually in *summary* and *description* fields but also in the categorial fields such as *product*, *component*, *priority*, *etc.* To capture this observation, given a bug report d and a query bug report q, our retrieval function $REP(d,q)$ is a linear combination of seven features, in the following form where w_i is the weight for the i-th feature $feature_i$.

$$REP(d,q) = \sum_{i=1}^{7} w_i \times feature_i \tag{6}$$

Each weight represents the degree of importance of its corresponding feature. If a feature is good at distinguishing similar reports from dissimilar ones, its weight should be larger than those features with weaker distinguishing power. Figure 2 shows the definitions of the seven features, which can be classified into two types:

Textual Features. The first feature defined in (7) is the textual similarity between two bug reports over the fields *summary* and *description* computed by $BM25F_{ext}$. The second feature

defined in (8) is the same as the first one, except that the fields *summary* and *description* are represented in bigrams. (A bigram consists of two consecutive words.)

Categorial Features. The rest five features are categorial features: the features 3–5 are based on the equality of the fields *product*, *component* and *type*; whereas the sixth and seventh features are the reciprocal of the distance between two *priorities* or *versions* respectively.

$$feature_1(d,q) = BM25F_{ext}(d,q) \text{ //of unigrams} \quad (7)$$

$$feature_2(d,q) = BM25F_{ext}(d,q) \text{ //of bigrams} \quad (8)$$

$$feature_3(d,q) = \begin{cases} 1, & \text{if } d.prod = q.prod \\ 0, & \text{otherwise} \end{cases} \quad (9)$$

$$feature_4(d,q) = \begin{cases} 1, & \text{if } d.comp = q.comp \\ 0, & \text{otherwise} \end{cases} \quad (10)$$

$$feature_5(d,q) = \begin{cases} 1, & \text{if } d.type = q.type \\ 0, & \text{otherwise} \end{cases} \quad (11)$$

$$feature_6(d,q) = \frac{1}{1+|d.prio - q.prio|} \quad (12)$$

$$feature_7(d,q) = \frac{1}{1+|d.vers - q.vers|} \quad (13)$$

Fig. 2. Features in the Retrieval Function

The retrieval function REP in (6) has 19 free parameters in total. For the first and second features, as we compute similarities over *two* fields *summary* and *description*, (namely $K = 2$ in $BM25F_{ext}$), each feature has $(2 + 2 \times 2) = 6$ free parameters. Also, we have a weight for each of the 7 features in (6), thus overall REP has $(2 \times 6 + 7) = 19$ free parameters. Table III lists all these parameters. The next section discusses how we tune these parameters for REP to gain good retrieval performance for a certain bug repository.

C. Optimizing REP with Gradient Descent

The parameter tuning for REP is based on gradient descent. In this work, we do not follow our previous approach using linear kernel SVM to optimize REP, although REP is a linear combination of features. The reason is that SVM is able to infer the feature weights w_1–w_7 but cannot tune the free parameters inside the first and second features which are based on $BM25F_{ext}$, such as k_1 and k_3. However, similar to SVM, the optimization technique used in this paper is also a discriminative approach, as parameters are tuned by contrasting similar pairs of reports against dissimilar ones.

Performing gradient descent also needs a training set. In the following, we first discuss how to construct a training set in a suitable format for our algorithm from a set of bug reports, and then detail our training algorithm.

1) Creating Training Set: As mentioned in Section II-D, the training set is a set of training instances of the form (q, rel, irr), where q is a query bug report, rel is a duplicate report of q, and irr is a report on a different bug.

TABLE III
PARAMETERS IN REP

Param	Description	Init	Example
w_1	weight of $feature_1$ (unigram)	0.9	1.163
w_2	weight of $feature_2$ (bigram)	0.2	0.013
w_3	weight of $feature_3$ (product)	2	2.285
w_4	weight of $feature_4$ (component)	0	0.032
w_5	weight of $feature_5$ (report type)	0.7	0.772
w_6	weight of $feature_6$ (priority)	0	0.381
w_7	weight of $feature_7$ (version)	0	2.427
$w_{summ}^{unigram}$	weight of *summary* in $feature_1$	3	2.980
$w_{desc}^{unigram}$	weight of *description* in $feature_1$	1	0.287
$b_{summ}^{unigram}$	b of *summary* in $feature_1$	0.5	0.703
$b_{desc}^{unigram}$	b of *description* in $feature_1$	1	1.000
$k_1^{unigram}$	k_1 in $feature_1$	2	2.000
$k_3^{unigram}$	k_3 in $feature_1$	0	0.382
w_{summ}^{bigram}	weight of *summary* in $feature_2$	3	2.999
w_{desc}^{bigram}	weight of *description* in $feature_2$	1	0.994
b_{summ}^{bigram}	b of *summary* in $feature_2$	0.5	0.504
b_{desc}^{bigram}	b of *description* in $feature_2$	1	1.000
k_1^{bigram}	k_1 in $feature_2$	2	2.000
k_3^{bigram}	k_3 in $feature_2$	0	0.001

Algorithm 2 Constructing a Training Set from a Repository

```
1:  TS = ∅: resultant training set
2:  N > 0: parameter controlling the size of TS
3:  for each bucket B in the repository do
4:      R = {B.master} ∪ B.duplicates
5:      for each report q in R do
6:          for each report rel in R–{q} do
7:              for i = 1 to N do
8:                  randomly choose a report irr s.t. irr ∉ R
9:                  TS = TS ∪ {(q, rel, irr)}
10:             end for
11:         end for
12:     end for
13: end for
14: return TS
```

Algorithm 2 lists the procedure to construct a training set from a repository. Generally, in each bucket of the repository, it pairs two reports as a relevant query-document pair (q, rel) at line 5 and 6. Next in line 7–10, it randomly chooses N reports which are not duplicate of the current bucket as irrelevant documents, and lastly pairs the relevant pair against each of the N reports to form N training instances. In our case studies, we set N to 30.

2) Parameter Tuning: To apply gradient descent, for each parameter x in REP, we manually derive $\frac{\partial RNC}{\partial x}$ – the partial derivativeof RNC with respect to x.

Next, we initialize each parameter with a default value. For parameters in $BM25F_{ext}$, we use recommended default values for parameters of $BM25F$ from information retrieval area. For example, for a lengthy field, its b should be big to perform length normalization, whereas for a short filed its b

is supposed to be small. Thus we instantiate $b_{summ} = 0.5$ and $b_{desc} = 1$. Based on our previous experience in [5], the terms in *summary* are usually three times as important as those in *description*, and hence we initially set $w_{summ} = 3$ and $w_{desc} = 1$. The column *Init* in Table III shows these initial values used in our case studies. Given a training set, we start the tuning by calling the procedure in Algorithm 1. The column *Example* in Table III displays a set of example values we get after tuning for OpenOffice. The tuned value for a parameter may vary with a different initial value, but it does not affect the retrieval performance much as the tuning algorithm coordinates and adjusts all the free parameters towards the optimal. Algorithm 3 lists the optimization process in detail.

Algorithm 3 Tuning Parameters in REP

1: initialize free parameters in REP with default values
2: fix k_1 and set $k_1 = 2$ in $feature_1$ and $feature_2$
3: fix k_3 and set $k_3 = 0$ in $feature_1$ and $feature_2$ {tune unfixed parameters: w_1–w_7, b, w_{summ} and w_{desc}}
4: call Algorithm 1 with $N = 24$ and $\eta = 0.001$
5: unfix k_3, fix b, w_{summ} and w_{desc} in $feature_1$ and $feature_2$ {tune unfixed parameters: w_1–w_7, k_3.}
6: call Algorithm 1 with $N = 24$ and $\eta = 0.001$ again

Following the analysis in [8] [10], tuning the weights of *summary* and *description* fields is redundant to tuning k_1. Thus, we fix k_1 to an arbitrary value $i.e.$, 2 and let the tuning algorithm tune the remaining parameters for $k_1 = 2$. Different from $BM25F$, $BM25F_{ext}$ has one more parameter k_3 that determines the scaling of term weights in queries. Tuning it is also redundant to tuning the weights of *summary* and *description* to some extent. Hence instead of tuning all these parameters together, we tune the parameters in two rounds:

1) In the first round in line 3–4, we fix k_3 in $feature_1$ and $feature_2$ to 0, and tune the other parameters.
2) In the second round in line 5–6, we unfix k_3, fix all the other parameters in $feature_1$ and $feature_2$ to their current values, and start tuning the rest of unfixed parameters including w_1–w_7 and k_3.

In addition to avoiding redundant tuning, we enjoy another benefit from the two-round tuning. As we fix k_3 to 0 in the first round, the $BM25F_{ext}$ in $feature_1$ and $feature_2$ is reduced to $BM25F$. All its partial derivatives except the one with respect to k_3 become equal to those of $BM25F$ respectively. Therefore in our implementation, we use the ones of $BM25F$ instead of $BM25F_{ext}$ in the first round, as they are faster and much simpler.

IV. Case Studies

We have built a prototype[2] to validate the effectiveness of the extension to $BM25F$ and our new retrieval function, and have applied it to the bug repositories of three large open source projects, OpenOffice, Mozilla and Eclipse. The experiments simulate the real-world bug triaging process, that is, for each duplicate report, we use the proposed techniques to retrieve a list of top similar master reports from the bug repository. The evaluation of the retrieval performance is measured by two metrics, *recall rate@k* and *mean average precision* (MAP). By fixing the size of the top list to k, *recall rate@k* defined in (14) measures the fraction of duplicate reports whose masters are successfully detected in the retrieved top-k masters ($N_{detected}$), among all the duplicate reports (N_{total}) used in testing the retrieval process.

$$recall\ rate@k = \frac{N_{detected}}{N_{total}} \tag{14}$$

MAP is a single-figure measure of ranked retrieval results independent of the size of the top list. It is designed for general ranked retrieval problem, where a query can have multiple relevant documents. However, duplicate bug report retrieval is special as each query (duplicate report) has only one relevant document (master report), thus MAP in our case studies is reduced to the simplified form in (15). For the complete form, please refer to [14]. Given a set Q of duplicate reports, for each duplicate, the system continually retrieves masters in descendent order of similarity until the right master is retrieved, and records its index in the ranked list. Then MAP can be computed as follows,

$$MAP(Q) = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{index_i} \tag{15}$$

where $index_i$ is the index where the right master is retrieved for the i-th query. Higher MAP means that for each duplicate the retrieval system can return the right master at a higher place in the ranked result list, saving triagers' time on checking if a new report is a duplicate, and on finding its associated master report.

Our case studies serve as two purposes: the first is to validate the effectiveness of $BM25F_{ext}$ over $BM25F$; the second is to compare the retrieval performance of the proposed retrieval function REP, our previous retrieval model based on SVM [5], and the work by Sureka and Jalote in [6].

A. Experimental Setup

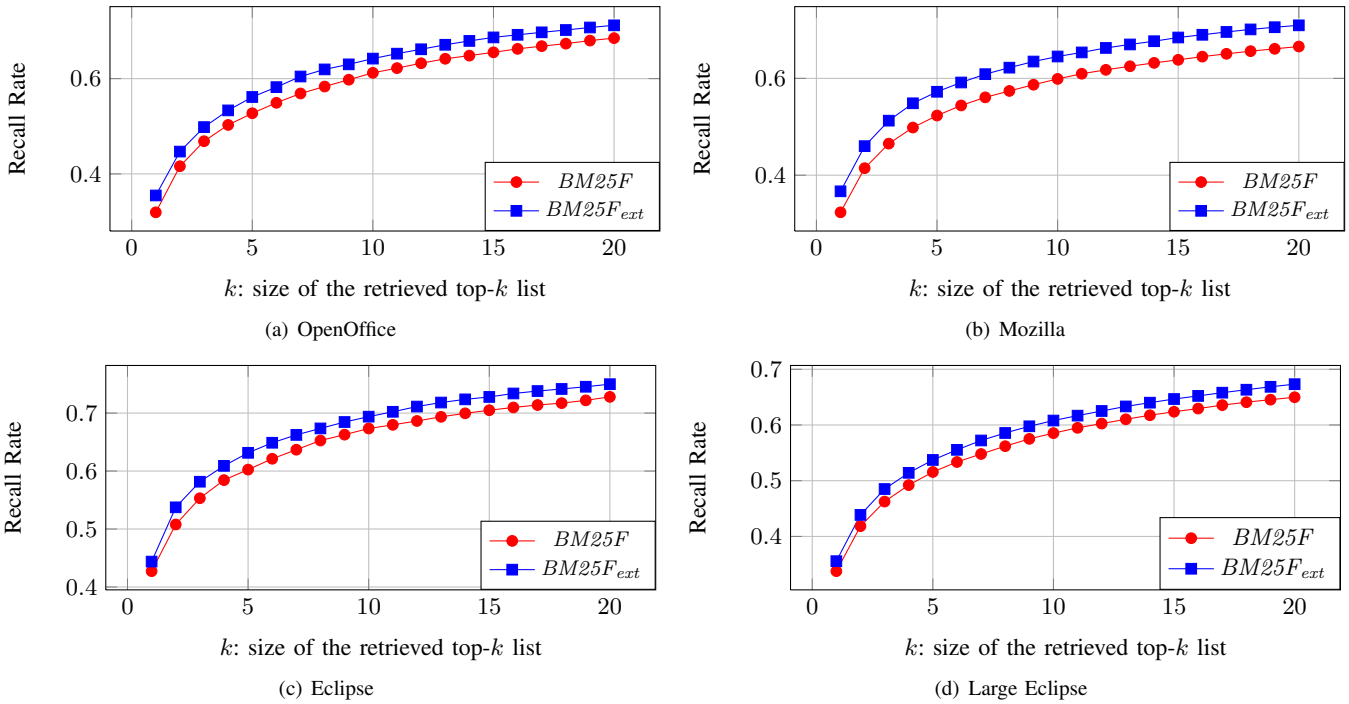
We used the bug repositories of three large open source projects: OpenOffice, Mozilla and Eclipse. OpenOffice is an open source counterpart of Microsoft Office. Mozilla is a community hosting multiple open source projects such as the famous web browser Firefox, email client Thunderbird. Eclipse is an extensible multi-language software development environment written in Java. These three projects are diverse in terms of purposes, users and implementation languages, thus help generalizing the conclusions of the experiments on them.

We extracted four report datasets from them by choosing reports submitted within a period of time T. In particular, we created two datasets from Eclipse: one is for reports submitted in 2008, whereas the other is for reports submitted from the

[2]Both implementation and dataset are available online at (http://www.comp.nus.edu.sg/~suncn/ase11/)

TABLE IV
DETAILS OF DATASETS

Dataset	Size	Period		Training Reports		Testing Reports	
		From	To	#Duplicate	#All	#Duplicate	#All
OpenOffice	31,138	2008-01-01	2010-12-21	200	3,696	3,171	27,442
Mozilla	75,653	2010-01-01	2010-12-31	200	4,529	6,725	71,124
Eclipse	45,234	2008-01-01	2008-12-31	200	5,863	2,880	39,371
Large Eclipse	209,058	2001-10-10	2007-12-14	200	3,528	27,295	205,530



(a) OpenOffice (b) Mozilla

(c) Eclipse (d) Large Eclipse

Fig. 3. Effectiveness of $BM25F_{ext}$ Compared to $BM25F$ in *Recall Rate*

beginning of Eclipse project to 2007. The reason to create the latter larger one[3] is that Sureka and Jalote used this dataset in their work [6] and we intend to compare with theirs using the same benchmark. Furthermore, this dataset spans a long time and is extremely large, hence it can further validate our technique across a long period of time.

These reports include all defect reports, maintenance tasks and feature requests. Each duplicate report has been labeled as *duplicate* by triagers and linked to its *master*. This information is used to test the retrieval and to measure the performance, *i.e.,* extracting duplicates to construct a training set, picking a duplicate to retrieve a top-k similar masters, and determining whether a master is the correct master of a duplicate. Since our retrieval function ($i.e.$, (6)) involves the order of versions, we manually recovered the chronological order of all the versions for OpenOffice by searching the release date of each version on the internet. We did not do this for the other datasets due to difficulty but we believe that it should be easy for project members to get and maintain such version order. For the other datasets, we just assume that all reports have the same version.

[3]We download this dataset from (http://msr.uwaterloo.ca/msr2008/challenge/)

Table IV details the four datasets. Our case studies follow the approach used in past studies on duplicate bug report retrieval [2]–[5]. We selected the *first* M reports in the repository (based on chronological order) of which 200 reports are duplicates as training set to tune the parameters in the retrieval function REP. The column *Training Reports* displays the ratio of duplicates and all reports in the training set. Besides serving as a training set in our approach, those M reports are also used to simulate the initial bug repository for all experimental runs. The rest of the duplicates are used for testing the retrieval process, shown in column *Testing Reports*. At each experimental run, we iterate through the testing reports in chronological order. Once reaching a duplicate report R, we apply the corresponding technique to retrieve R's potential master reports until its right master is received. After each retrieval is done, we record the index of R's master in the top list for *recall rate@k* and MAP calculation, and then add R to the repository. After the last iteration is done, the *recall rate* for different sizes of the top list and MAP are calculated.

The prototype was implemented in C++, and all experiments were carried on a Linux PC with Intel Core 2 Quad CPU 3.0GHz and 8GB memory. Since the training set is constructed randomly ($c.f.$ Algorithm 2), and the gradient descent also

involves randomness (*c.f.* Algorithm 1), we repeated all experiments five times and take the average values for our analysis and conclusion.

B. Effectiveness of $BM25F_{ext}$

In this experiment, we aim to validate the effectiveness of $BM25F_{ext}$ over $BM25F$. To ensure fairness, we only involve textual similarity on *summary* and *description* in our experiment, and omit the use of any categorial features. Figure 3 shows the curve of the *recall rate@k* of the two similarity measures on the four datasets. On each dataset for each different size of the top-k result list, $BM25F_{ext}$ performs constantly better than $BM25F$ and it gains 4%–11%, 7%–13%, 3%–6% and 3%–5% relative improvement over $BM25F$ on OpenOffice, Mozilla, Eclipse and Large Eclipse datasets respectively. Moreover, Table V shows the mean average precision of the two metrics, and the last row is the relative improvement by $BM25F_{ext}$ over $BM25F$, which is up to 10.68%. Based on the large set of testing query duplicates – 40,071 in total, we conclude that in the context of duplicate bug report retrieval, $BM25F_{ext}$ is more effective than $BM25F$.

TABLE V
MAP OF $BM25F_{ext}$ AND $BM25F$

	OpenOffice	Mozilla	Eclipse	Large Eclipse
$BM25F_{ext}$	45.21%	46.22%	53.21%	44.22%
$BM25F$	41.92%	41.76%	51.06%	42.25%
Relative Impro.	7.86%	10.68%	4.20%	4.66%

C. Effectiveness of Retrieval Function

We evaluate the performance of our new retrieval function REP against previous techniques, including our previous work based on support vector machine [5] and a recent work using character n-gram-based features by Sureka and Jalote in [6]. We apply both our previous technique and REP on the four datasets and compare them directly. However, due to high overhead, our previous technique on Large Eclipse is not able to complete despite running for two days, thus there is no result for this experimental run. For the work by Sureka and Jalote, they do the evaluation on the Large Eclipse dataset. We compare our result with the accuracy results reported in their paper.

Figure 4 shows the *recall rate* of REP and our previous technique on the four datasets. In the figure, *SVM* represents our previous technique, *REP-V* stands for the REP retrieval function with version information considered, and *REP-NV* is the REP without version information considered. As mentioned before, we only recovered the chronological order of versions for OpenOffice, therefore only applied *REP-V* to OpenOffice. From Figure 4(a), we can see that *REP-V* can improve the recall rate over *REP-NV* by 2–4%. In Figure 4(d), only the performance of *REP-NV* is shown, as running of *SVM* experiences time-out. Overall, the new retrieval function outperforms *SVM* 14–27% in OpenOffice dataset, 10–26% in Mozilla dataset and 12–22% in Eclipse dataset. Table VI displays the MAP of each experimental run. The last row is the relative improvement brought by REP over *SVM*.

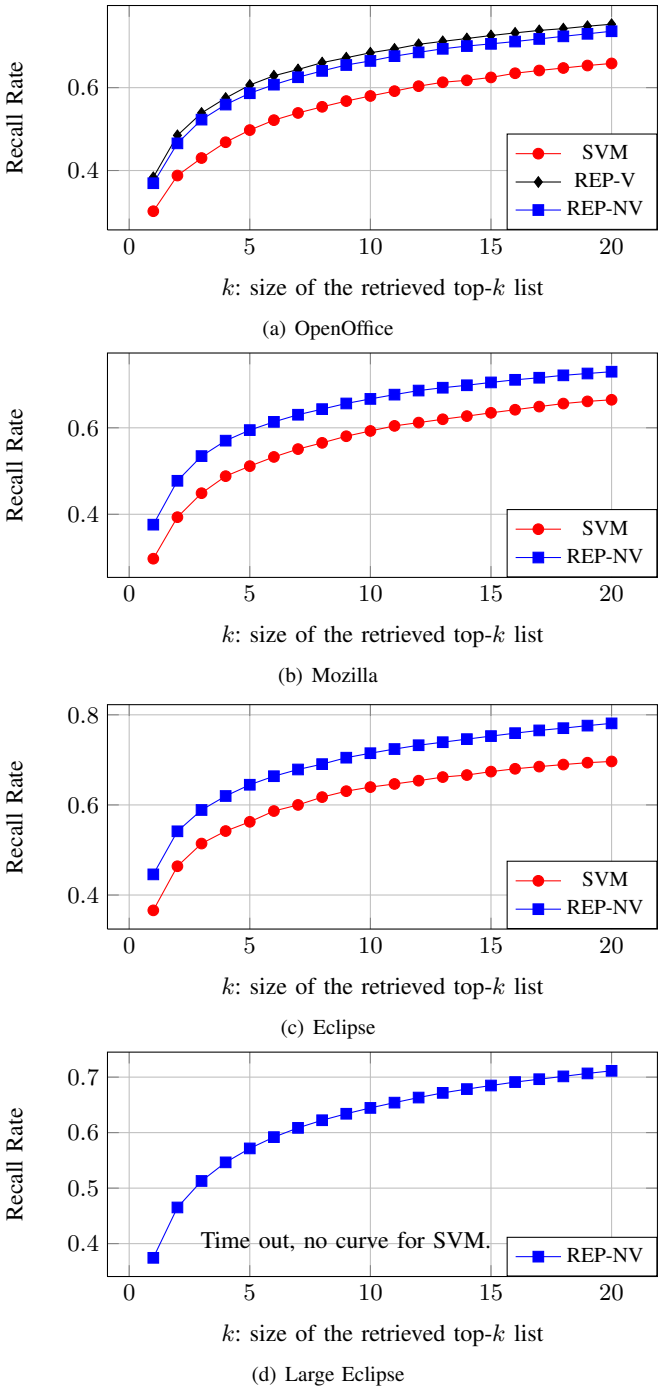


Fig. 4. Comparison with Our Previous Approach

In [6], Sureka and Jalote randomly selected 1100 duplicate reports to test their approach and reported that *recall rate* @10 was 21%, @20 was 25% and @2000 was 68%. We experimented on the same dataset, Large Eclipse, and we tested all the 27,295 duplicate reports to avoid randomness,

TABLE VI
MAP OF *REP-V REP-NV* AND *SVM*

	OpenOffice	Mozilla	Eclipse	Large Eclipse
REP-V	48.63%	–	–	–
REP-NV	47.03%	47.71%	53.96%	46.73%
SVM	39.54%	39.79%	46.09%	–
Relative Impro.	22.99%	19.91%	17.20%	–

which we believe can produce more reliable result. The *recall rate* @1 is 37% and @20 is 71%. We can thus safely conclude that our technique improves upon past techniques in recall rate@k

Aside from the performance improvement, REP also significantly reduces the runtime compared to our past approach that uses SVM [5]. Table VII displays the time needed to finish an experiment run for each dataset.

TABLE VII
OVERHEAD OF *SVM* AND *REP* (IN SECONDS)

	OpenOffice	Mozilla	Eclipse	Large Eclipse
REP	139	1603	277	7037
SVM	3408	19411	4267	> 2 days

V. RELATED WORK

In this section, we describe the existing studies on duplicate bug report detection and other studies that analyze bug reports.

A. Duplicate Bug Report Detection

There have been a number of past studies that detect duplicate bug reports. We would present them in chronological order and then elaborate the difference between our work and theirs.

Runeson et al. perform one of the first studies on finding duplicate bug reports [3]. They take natural language text of bug reports and perform standard tokenization, stemming, and stop word removal. Each bug report is then characterized as a bag of word tokens and is modeled as a feature vector, where each feature corresponds to a word in the bug report. The feature value in the vector is computed based on the following formula: $1 + log_2(TF(word))$, while TF is the term frequency of the word. Each bug report is then compared to other bug reports by computing the similarity of their corresponding feature vectors. Three similarity measures are analyzed: cosine, dice, and jaccard. Bug reports with high similarity scores are likely to be duplicates. Their study on bug reports of Sony Ericsson Mobile Communications show that cosine performs best and is able to detect 40% of the duplicate bug reports.

Wang et al. use another feature vector construction approach where each feature is computed by considering both term frequency (TF) and inverse document frequency (IDF) of the words in the bug reports following the formula: $TF(word) * IDF(word)$ [4]. Furthermore, they consider an additional source of information to measure the similarity between two bug reports namely program execution traces. Using cosine similarity of the composite feature vectors, they detect top-k similar reports as candidate duplicate bug reports. Their approach could detect 67%–93% of the duplicate bug reports in their case study on a small subset of Mozilla Firefox bug reports. They only consider a relatively small case study as bug reports which contain execution traces are scarce.

Jalbert and Weimer propose another feature vector construction approach where each feature is computed by the following formula: $3+2*log_2(TF(word))$ [2]. Cosine similarity is also used to extract top-k similar reports as candidate duplicate bug reports. They also propose a classification technique to detect bug reports that are duplicated.

Sun et al. propose to use a discriminative approach to detect bug reports. A Support Vector Machine (SVM) is used to train a model that would compute the probability of two reports being duplicate [5]. This probability is used to detect top-k most similar reports as candidate duplicate bug reports. Their case study on bug reports from OpenOffice, Firefox, and Eclipse, show that they could outperform the approaches proposed in [2]–[4] when only the natural language text of the bug reports is considered.

More recently, Sureka and Jalote propose an approach that consider not word tokens but n-grams as features in a feature vector that characterizes a bug report [6]. They show that their approach is able to detect duplicate bug reports with reasonable accuracy on a large bug report corpus from Eclipse. No comparative study however is performed to compare their approach with the other existing approaches.

Similarities and Differences. We address the same problem of detecting duplicate bug reports or more accurately retrieving top-k related bug reports from a collection of bug reports. We consider natural language text of the bug reports, similar to the work in [2]–[6]. Aside from natural language text, we also consider other categorial features available in Bugzilla. The study by Jalbert and Weimer also considers categorial features [2]. Different from the work by Wang et al. [4], we ignore execution traces as these are often not available in bug reports. Indeed, out of the many reports that we study only a small proportion of them contain execution trace information. In this study, we have extended one of the latest textual similarity measures in information retrieval for retrieving structured documents namely $BM25F$. Lastly we show that our proposed retrieval function involving textual and categorial similarities outperforms other state-of-the-art measures proposed in the literature, $i.e.$, [5], [6].

B. Other Bug Report Related Studies

Categorization of bug reports is investigated by Anvik et al. [15], Cubranic and Murphy [16], Pordguski et al [17], and Francis et al. [18]. In [15], [16], this categorization is used to assign bug reports to the right developers. A study on predicting the severity of bug reports is performed by Menzies and Marcus [19]. Bettenburg et al. extract stack traces, codes, and patches from textual bug reports [20]. Ko and Myers

investigate the differences between defect reports and feature requests [21].

Anvik et al. perform an empirical study on the characteristics of bug repositories including the number of reports that a person submitted and the proportion of different resolutions [1]. Sandusky et al. study the statistics of duplicate bug reports [22]. Hooimeijer and Weimer develop a model to predict bug report quality [23]. Bettenburg et al. survey developers of Eclipse, Mozilla, and Apache to study what developers care most in a bug report [24]. Bettenburg et al. later show that duplicate bug reports might be useful [7]. Duplicate bug reports could potentially provide different perspectives to the same defect potentially enabling developers to better fix the defect in a faster amount of time. Still there is a need to detect bug reports that are duplicate of one another.

VI. Conclusion & Future Work

In this work, we improve the accuracy of duplicate bug retrieval in two ways. First, $BM25F$ is an effective textual similarity measure which is originally designed for short unstructured queries, and we extend it to $BM25F_{ext}$ specially for lengthy structured report queries by considering weight of terms in queries. Second, we propose a new retrieval function REP fully utilizing not only text but also other information available in reports such as *product*, *component*, *priority* $etc.$ A two-round gradient descent contrasting similar pairs of reports against dissimilar ones, is adopted to optimize REP based on a training set.

We have investigated the utility of our technique on 4 sizable bug datasets extracted from 3 large open-source projects, $i.e.$, OpenOffice, Firefox and Eclipse; and find that both $BM25F_{ext}$ and REP are indeed able to improve the retrieval performance. Particularly, the experiments on the four datasets show that $BM25F_{ext}$ improves *recall rate@k* by 3–13% and MAP by 4–11% over $BM25F$. For retrieval performance of REP, compared to our previous work based on SVM, it increases *recall rate@k* by 10–27%, and MAP by 17–23%; compared to the work by Sureka and Jalote [6], REP performs with *recall rate@k* of 37–71% ($1 \leq k \leq 20$), and MAP of 46%, which are much higher than the results reported in their paper.

In the future, we plan to build indexing structure of bug report repository to speed up the retrieval process. Last but not least, we plan to integrate our technique into Bugzilla tracking system.

Acknowledgment

We are grateful to the reviewers for their valuable comments. This work is partially supported by a research grant R-252-000-403-112.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 35–39.

[2] N. Jalbert and W. Weimer, "Automated Duplicate Detection for Bug Tracking Systems," in *proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[3] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *proceedings of the International Conference on Software Engineering*, 2007.

[4] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information," in *proceedings of the International Conference on Software Engineering*, 2008.

[5] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010, pp. 45–56.

[6] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, 2010, pp. 366–374.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *ICSM08: Proceedings of IEEE International Conference on Software Maintenance*, 2008, pp. 337–345.

[8] S. Robertson, H. Zaragoza, and M. Taylor, "Simple BM25 Extension to Multiple Weighted Fields," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, 2004, pp. 42–49.

[9] H. Zaragoza, N. Craswell, M. J. Taylor, S. Saria, and S. E. Robertson, "Microsoft cambridge at trec 13: Web and hard tracks," in *TREC*, 2004.

[10] M. Taylor, H. Zaragoza, N. Craswell, S. Robertson, and C. Burges, "Optimisation methods for ranking functions with multiple parameters," in *Proceedings of the 15th ACM international conference on Information and knowledge management*, 2006, pp. 585–593.

[11] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to Rank Using Gradient Descent," in *Proceedings of the 22nd international conference on Machine learning*, 2005, pp. 89–96.

[12] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997, pp. 88–95.

[13] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008, pp. 232–233.

[14] ——, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008, pp. 158–163.

[15] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *proceedings of the International Conference on Software Engineering*, 2006.

[16] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004, pp. 92–97.

[17] 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*, 2003, pp. 465–475.

[18] P. Francis, D. Leon, and M. Minch, "Tree-based methods for classifying software failures," in *ISSRE*, 2004.

[19] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM08: Proceedings of IEEE International Conference on Software Maintenance*, 2008, pp. 346–355.

[20] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 27–30.

[21] A. Ko and B. Myers, "A linguistic analysis of how people describe software problems," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 127–134.

[22] R. J. Sandusky, L. Gasser, R. J. S, U. L. Gasser, and G. Ripoche, "Bug report networks: Varieties, strategies, and impacts in a f/oss development community," in *International Workshop on Mining Software Repositories*, 2004, pp. 80–84.

[23] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 34–43.

[24] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 308–318.