A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval

Chengnian Sun[1], David Lo[2], Xiaoyin Wang[3], Jing Jiang[2], Siau-Cheng Khoo[1]
[1]School of Computing, National University of Singapore
[2]School of Information Systems, Singapore Management University
[3]Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
suncn@comp.nus.edu.sg, davidlo@smu.edu.sg, wangxy06@sei.pku.edu.cn,
jingjiang@smu.edu.sg, khoosc@comp.nus.edu.sg

ABSTRACT

Bug repositories are usually maintained in software projects. Testers or users submit bug reports to identify various issues with systems. Sometimes two or more bug reports correspond to the same defect. To address the problem with duplicate bug reports, a person called a triager needs to manually label these bug reports as duplicates, and link them to their "master" reports for subsequent maintenance work. However, in practice there are considerable duplicate bug reports sent daily; requesting triagers to manually label these bugs could be highly time consuming.

To address this issue, recently, several techniques have be proposed using various similarity based metrics to detect candidate duplicate bug reports for manual verification. Automating triaging has been proved challenging as two reports of the same bug could be written in various ways. There is still much room for improvement in terms of accuracy of duplicate detection process. In this paper, we leverage recent advances on using discriminative models for information retrieval to detect duplicate bug reports more accurately. We have validated our approach on three large software bug repositories from Firefox, Eclipse, and OpenOffice. We show that our technique could result in 17–31%, 22–26%, and 35–43% relative improvement over state-of-the-art techniques in OpenOffice, Firefox, and Eclipse datasets respectively using commonly available natural language information only.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Management, Reliability

1. INTRODUCTION

ICSE'10, May 2–8, 2010, Cape Town, South Africa.

Due to complexities of systems built, software often comes with defects. Software defects have caused billions of dollars lost [20]. Fixing defects is one of the most frequent reasons for software maintenance activities which also goes to 70 billion US dollars in the United States alone [19].

In order to help track software defects and build more reliable systems, bug tracking tools have been introduced. Bug tracking systems like Bugzilla[1] enable many users to serve as "testers" and report their findings in a unified environment. These bug reports are then used to guide software corrective maintenance activities and result in more reliable software systems. Via the bug tracking systems, users are able to report new bugs, track statuses of bug reports, and comment on existing bug reports.

Despite the benefits of a bug reporting system, it does cause some challenges. As bug reporting process is often uncoordinated and ad-hoc, often the same bugs could be reported more than once by different users. Hence, there is often a need for manual inspection to detect whether the bug has been reported before. If the incoming bug report is not reported before then the bug should be assigned to a developer. However, if other users have reported the bug before then the bug would be classified as being a duplicate and attached to the original first-reported "master" bug report. This process referred to as triaging often takes much time. For example, for the Mozilla programmers, it has been reported in 2005 that "everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [2].

In order to alleviate the heavy burden of triagers, there have been recent techniques to automate the triaging process in two ways. The first one is automatically filtering duplicates to prevent multiple duplicate reports from reaching triagers [11]. The second is providing a list of similar bug reports to each incoming report under investigation [17, 22, 11]; with the help, rather than checking against the entire collection of bug reports, a triager could first inspect the top-k most similar bug reports returned by the systems. If there is a report in the list that reports about the same defect as the new one, then the one is a duplicate, The triager then marks it as a duplicate and adds a link between the two duplicates for subsequent maintenance work. In our paper, instead of filtering duplicates, we choose the second approach as duplicate bug reports are not necessarily bad. As stated in [3], one report usually does not carry enough information for developers to dig into the reported defect,

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

while duplicate reports can complement one another.

To achieve better automation and thus save triagers' time, it is important to improve the quality of the ranked list of similar bug reports given a new bug report. There have been several studies on retrieving similar bug reports. However, the performance of these systems is still relative low, making it hard to apply them in practice. The low performance is partly due to the following limitations of the current methods. First, all the three techniques in [17, 22, 11] employ one or two features to describe the similarity between reports, despite the fact that other features are also available for effective measurement of similarity. Second, different features contribute differently towards determining similarities. For example, the feature capturing similarity between summaries of two reports are more effective than that between descriptions, as summaries typically carry more concise information. However, as project contexts evolve, the relative importance of features might vary. This can cause the past techniques, which are largely based on absolute rating of importance, to deteriorate in their performance.

More accurate results would mean more automation and less effort by triagers to find duplicate bug reports. To address this need, we propose a *discriminative model based approach* that further improves accuracy in retrieving duplicate bug reports by up to 43% on real bug report datasets.

Different from the previous approaches that rank similar bug reports based on similarity score of vector space representation, we develop a discriminative model to retrieve similar bug reports from a bug repository. We make use of the recent advances in information retrieval community that uses a classifier to retrieve similar documents from a collection[15]. We build a model that contrasts duplicate bug reports from non-duplicate bug reports and utilize this model to extract similar bug reports, given a query bug report under consideration.

We strengthen the effectiveness of bug report retrieval system by introducing many more relevant features to capture the similarity between bug reports. Moreover, with the adoption of the discriminative model approach, the relative importance of each feature will be automatically determined by the model through assignment of an optimum weight. Consequently, as bug repository evolves, our discriminative model also evolves to guarantee the all the weights remain optimum at all time. In this sense, our process is more adaptive, robust, and automated.

We evaluate our discriminative model approach on three large bug report datasets from large programs including Firefox, an open source web browser, Eclipse, a popular open source integrated development environment, and OpenOffice, a well-known open source rich text editor. In terms of the range of types of programs considered for evaluation, to the best of our knowledge, we are the first to investigate the applicability of the approach on different types of systems. We show that our technique could result in 17–31%, 22–26%, and 35–43% improvement over state-of-the-art techniques [17, 22, 11] in OpenOffice, Firefox, and Eclipse datasets respectively using commonly available natural language information alone.

We summarize our contributions as follows:

1. We employ in total 54 features to comprehensively evaluate the similarity between two reports.
2. We propose a discriminative model based solution to retrieve similar bug reports from a bug tracking system. Our model can automatically assign optimum weight to each feature and evolve along with the changes of bug repositories.
3. We are the first to analyze the applicability of duplicate bug report detection techniques across different sizable bug repositories of various large open source programs including OpenOffice, Firefox, and Eclipse.
4. We improve the accuracy of state-of-the-art automated duplicate bug detection systems by up to 43% on different open-source datasets.

The paper is organized as follows. Section 2 presents some background information on bug reports, information retrieval, and discriminative model construction. Section 3 presents our approach to retrieving similar bug reports for duplicate bug report detection. Section 4 describes our case study on sizable bug repositories of different open source projects and shows the utility of the proposed approach in improving the state-of-the-art detection performance. Section 5 discusses some important consideration about our approach. Section 6 discusses related work, and finally, Section 7 concludes and describes some potential future work.

2. BACKGROUND

In general, duplicate bug report retrieval involves information extraction from and comparison between documents in natural language. This section covers the necessary background and foundation techniques to perform the task in our approach.

2.1 Duplicate Bug Reports

A bug report is a structured record consisting of several fields. Commonly, they include *s*ummary, *d*escription, *p*roject, *s*ubmitter, *p*riority and so forth. Each field carries a different type of information. For example, *s*ummary is a concise description of the defect problem while *d*escription is the detailed outline of what went wrong and how it happened. Both of them are in natural language format. Other fields such as *p*roject, *p*riority try to characterize the defect from other perspectives.

In a typical software development process, the bug tracking system is open for testers or even for all end users, so it is unavoidable that two people may submit different reports on the same bug. This causes the problem of duplicate bug reports. As mentioned in [17], duplicate reports can be divided into two categories. One describes the same failure, and the other depicts two different failures both originated from the same root cause. In this paper, we only handle the first category. As an example, Table 1 shows three pairs of duplicate reports extracted from Issue Tracker of OpenOffice. Only the summaries are listed.

Usually, new bug reports are continually submitted. When triagers identify that a new report is a duplicate of an old one, the new one is marked as duplicate. As a result, given a set of reports on the same defect, only the oldest one in the set is not marked as duplicate. We refer to the oldest one as *master* and the others as its *duplicates*.

A bug repository could be viewed as containing two groups of reports: masters and duplicates. Since each duplicate must have a corresponding master and both reports are on the same defect, the defects represented by all the duplicates

Table 1: Examples of Duplicate Bug Reports

ID	Summary
85064	[Notes2] No Scrolling of document content by use of the mouse wheel
85377	[CWS notes2] unable to scroll in a note with the mouse wheel
85502	Alt+<letter> does not work in dialogs
85819	Alt-<key> no longer works as expected
85487	connectivity: evoab2 needs to be changed to build against changed api
85496	connectivity fails to build (evoab2) in m4

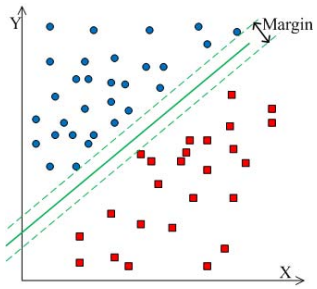


Figure 1: Maximum-Margin Hyperplane Calculated by SVM in Two-Dimensional Space

in the repository belong to the set of the defects represented by all the masters. Furthermore, typically each master report represents a distinct defect.

2.2 Information Retrieval

Information retrieval (IR) aims to extract useful information from unstructured documents, most of which are expressed in natural language. IR methods typically treat documents as "bags of words" and subsequently represent them in a high-dimensional vector space where each dimension corresponds to a unique word or term. In this paper, we use term and word interchangeably. The following describes some commonly used strategies to pre-process documents and methods to weigh terms.

Pre-processing. In order to computerize retrieval task, a sequence of actions should be taken first to preprocess documents using natural language processing techniques. Usually, this sequence comprises tokenization, stemming and stop word removal. A word token is a maximum sequence of consecutive characters without any delimiters. A delimiter in turn could be a space, punctuation mark, etc. Tokenization is the process of parsing a character stream into a sequence of word tokens by splitting the stream by the delimiters. Stemming is the process to reduce words to their *ground* forms. The motivation to do so is that different forms of words derived from the same root usually have similar meanings. By stemming, computers can capture this similarity via direct string equivalence. For example, a stemmer can reduce both "tested" and "testing" to "test". The last action is stop word removal. Stop words are those words carrying little helpful information for information retrieval task. These include pronouns such as "it", "he" and "she", link verbs such as "is", "am" and "are", etc. In our stop word list, in addition to removing 30 common stop words, we also drop common abbreviations such as "I'm", "that's", "we'll", etc.

Term-weighting. TF-IDF (Term Frequency-Inverse Document Frequency) is a common term-weighting scheme. It is a statistical approach to evaluating the importance of a term in a corpus. TF is a *local* importance measure. Given a term and a document, in general, TF corresponds to the number of times the term appears within the document. Different from TF, IDF is a *global* importance measure most commonly calculated by the formula within the corpus,

$$idf(term) = log_2(\frac{D_{all}}{D_{term}}) \quad (1)$$

In (1), D_{all} is the number of the documents in the corpus while D_{term} is the number of documents containing the term. Given a term, the fewer documents it is contained in, the more important it becomes. In our approach, we employ an *idf*-based formula to weigh terms.

2.3 Building Discriminative Models via SVM

Support Vector Machine (SVM) is an approach to building a discriminative model or classifier based on a set of labeled vectors. Given a set of vectors, some belonging to a positive class and others belonging to a negative class, SVM tries to build a hyperplane that separates vectors belonging to the positive class from those of the negative class with the largest margin. Figure 1 shows such kind of a hyperplane built by SVM with the maximum margin in a two-dimensional space. The resultant model could then be used to classify other unknown data points in vector representation and label them as either positive or negative. In this study, we use libsvm [6], a popular implementation of SVM.

3. OUR APPROACH

Duplicate bug report retrieval can be viewed as an application of information retrieval (IR) technique to the domain of software engineering, with the objective of improving productivity of software maintenance. In classical retrieval problem, the user gives a query expressing the information he/she is looking for. The IR system would then return a list of documents relevant to the query. For duplicate report retrieval problem, the triager receives a new report and inputs it to the duplicate report retrieval system as a query. The system then returns a list of potential duplicate reports. The list should be sorted in a descending order of relevance to the queried bug report.

Our approach adopts recent development on discriminative models for information retrieval to retrieve duplicate bug reports. Adapted from [15], we consider duplicate bug report retrieval as a binary classification problem, that is, given a new report, the retrieval process is to classify all existing reports into two classes: duplicate and non-duplicate. We compute 54 types of textual similarities between reports and use them as features for training and classification purpose.

The rest of this section is structured as follows: Subsection 3.1 gives a bird's eye view of the overall framework. Sub-section 3.2 explains how existing bug reports in the repository are organized. Sub-section 3.3 elaborates on how a discriminative model is built. Sub-section 3.4 describes how the model is applied for retrieving duplicate bug reports. Finally, Sub-section 3.5 describes how the model is updated when new triaged bug reports arrive.

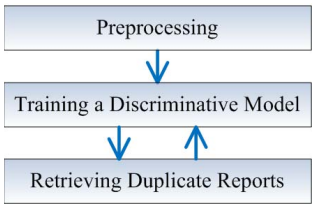


Figure 2: Overall Framework to Retrieve Duplicate Bug Reports

Buckets
Master-1: dup-1.1, dup-2.1, ……
Master-2: dup-2.1, dup-2.2, ……
Master-3: dup-3.1, dup-3.2, …...
……
Master-M: dup-M.1, dup-M.2, ……

Figure 3: Bucket Structure

3.1 Overall Framework

Figure 2 shows the overall framework of our approach. In general, there are three main steps in the system, *preprocessing*, *training a discriminative model* and *retrieving duplicate bug reports*.

The first step, preprocessing, follows a standard natural language processing style – tokenization, stemming and stop words removal – described in Sub-section 2.2. The second step, *training a discriminative model*, trains a classifier to answer the question "How likely are two bug reports duplicates of each other?". The third step, *retrieving duplicate bug reports*, makes use of this classifier to retrieve relevant bug reports from the repository.

3.2 Data Structure

All the reports in the repository are organized into a bucket structure. The bucket structure is a hash-map-like data structure. Each bucket contains a master report as the key and all the duplicates of the master as its value. As explained in Sub-section 2.1, different masters report different defects while a master and its duplicates report the same defect. Therefore, each bucket stands for a distinct defect, while all the reports in a bucket correspond to the same defect. The structure of the bucket is shown diagrammatically in Figure 3. New reports will also be added to the structure after they are labeled as duplicate or non-duplicate by triagers. If a new report is a duplicate, it will go to the bucket indexed by its master; otherwise, a new bucket will be created to include the new report and it becomes a master.

3.3 Training a Discriminative Model

Given a set of bug reports classified into masters and duplicates, we would like to build a discriminative model or a classifier that answers the question: "How likely are two input bug reports duplicate of each other?". This question is essential in our retrieval system. As described in Subsection 3.4, the answer is a probability describing the likelihood of these two reports being duplicate of each other. When a new report comes, we ask the question for each pair between the new report and all the existing reports in the repository and then retrieve the duplicate reports based on the probability answers. To get the answer we follow a multi-step approach involving example creation, feature extraction, and discriminative model creation via Support Vector Machines (SVMs).

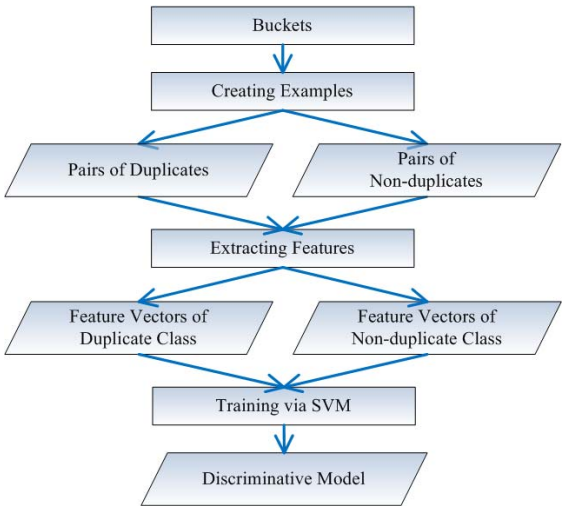


Figure 4: Training a Discriminative Model

The steps are shown in Figure 4. Based on the buckets containing masters associated with corresponding duplicates, we extract positive and negative examples. Positive examples correspond to pairs of bug reports that are duplicates of each other. Negative examples correspond to pairs of bug reports that are not duplicates of each other. Next, a feature extraction process is employed to extract features from the pairs of bug reports. These features must be rich enough to be able to discriminate between cases where bug reports are duplicate of one another and cases where they are distinct. These feature vectors corresponding to duplicates and non-duplicates are then input to an SVM learning algorithm to build a suitable discriminative model. The following sub-sections describe each of the steps in more detail.

3.3.1 *Creating Examples*

To create positive examples, for each bucket, we perform the following:

1. Create the pair (master, duplicate), where duplicate is one of the duplicates in the bucket and master is the original report in the bucket.
2. Create the pairs (duplicate_1,duplicate_2) where the two duplicates belong to the same bucket.

To create negative examples, one could pair one report from one bucket with another report from the other bucket. The number of negative examples could be much larger than the number of positive examples. As there are issues related to skewed or imbalanced dataset when building classification models (c.f. [13]), we choose to under-sample the negative examples, thus ensure that we have the same number of positive and negative examples.

At the end of the process, we have two sets of examples: one corresponds to examples of pairs of bug reports that are

duplicates, and the other corresponds to examples of pairs of bug reports that are non-duplicates.

3.3.2 Feature Engineering & Extraction

At times, limited features make it hard to differentiate between two contrasting datasets: in our case, pairs that are duplicates and pairs that are non-duplicates. Hence a rich enough feature set is needed to make duplicate bug report retrieval more accurate. Since we are extracting features corresponding to a pair of textual reports, various textual similarity measures between the two reports are good feature candidates. In our approach, we employ the following formula as the textual similarity.

$$sim(B_1, B_2) = \sum_{w \in B_1 \cap B_2} idf(w) \quad (2)$$

In (2), $sim(B_1, B_2)$ returns the similarity between two bags of words B_1 and B_2. The similarity is the sum of *idf* values of all the shared words between B_1 and B_2. The *idf* value for each word is computed based on a corpus formed from all the reports in the repository, which will be detailed further below. The rational why the similarity measure does not involve TF is that the measure with only IDF yields better performance indicated by Fisher score which will be detailed in Sub-section 5.2, and validated by the experiments.

Generally, each feature in our approach can then be abstracted by the following formula,

$$f(R_1, R_2) = sim(words\ from\ R_1, words\ from\ R_2) \quad (3)$$

From (3), a feature is actually the similarity between two bags of words from two reports R_1 and R_2.

One observation is that a bug report consists of two important fields: *s*ummary and *d*escription. So we can get three bags of words from one report, one bag from *s*ummary, one from *d*escription and one from *b*oth (summary+description). To extract a feature from a pair of bug reports, for example, one could compute the similarity between the bag of words from the *s*ummary of one report and the words from the *d*escription of the other. Alternatively, one could use the similarity between the words from *b*oth the summary and description of one report and those from the *s*ummary of the other. Other combinations are also possible.

Furthermore, we can compute three types of *idf*, as the bug repository can form three distinct corpora. One corpus is the collection of all the *s*ummaries, one corpus is the collection of all the *d*escriptions, and the other is the collection of all the *b*oth (summary+description). We denote the three types of *idf* computed within the three corpora by idf^{sum}, idf^{desc}, and idf^{both} respectively.

The output of function f defined in (3) depends on the choice of bag of words for R_1, the choice of bag of words for R_2 and the choice of *idf*. Considering each of the combinations as a separate feature, the total number of different features would be $3 \times 3 \times 3$, which is equal to 27. Figure 5 shows how the 27 features are extracted from a pair of bug reports.

Aside from considering words, we also consider bigrams – two consecutive words. With bigrams, considering different combinations of bag of words coming from and *idf* computed based on *s*ummaries,*d*escriptions, or *b*oth, we would have another 27 features which would then bring the number of features extracted to 54.

Algorithm 1 Calculate Candidate Reports for Q

Procedure ProposeCandidates
Input:
Q: a new report
Rep: the bug repository
N: the expected size of candidate list
Output:
$result$: a list of N masters of which Q is a likely duplicate
Body:
1: $Candidates$ = an empty min-heap of maximum size N
2: **for each** bucket $B \in Buckets(Rep)$ **do**
3: $similarity = PredictBucket(Q, B)$
4: $Master(B).similarity = similarity$
5: add $Master(B)$ to $Candidates$
6: **end for**
7: sort $Candidates$ in descending order of field $similarity$
8: **return** sorted $Candidates$

Algorithm 2 Calculate Similarity between Q and a Bucket

Procedure PredictBucket
Input:
Q: a new report
B: a bucket
Output:
max: the maximum similarity between Q and each report of B
Body:
1: $max = 0$
2: $tests = \{f54(Q, R) | R \in Reports(B)\}$
3: **for each** feature vector $t \in tests$ **do**
4: $probability = SVMPredict(t)$
5: $max = MAX(max, probability)$
6: **end for**
7: **return** max

3.3.3 Training Models

Before training, an extra action is taken to normalize the values of all features in the training set to within the range [-1,1]. This is to avoid the case that some features in the bigger range dominate those in the smaller range. We use a standard algorithm to normalize the feature vectors (c.f. [6]). The same normalization will also be applied when the resultant model is used for classifying unknown pairs.

We use libsvm [6] to train a discriminative model from the training set. As suggested by [15], we choose the linear kernel for SVM as it is efficient and effective in performing classification for information retrieval. A typical classifier would only give binary answers; in our case, whether two bug reports are duplicate or not. We are *not* interested in binary answers; rather, we are interested in knowing how likely two bug reports are duplicates. To do this we enable the probability estimation functionality of libsvm to train a discriminative model which is able to produce a probability of two bug reports being duplicates of each other.

3.4 Applying Models for Duplicate Detection

When a new report Q arrives, we can apply the trained model to retrieve a list of candidate reports of which Q is likely a duplicate. The retrieval details are displayed in Algorithm 1 and Algorithm 2.

Algorithm 1 returns a duplicate candidate list for a new

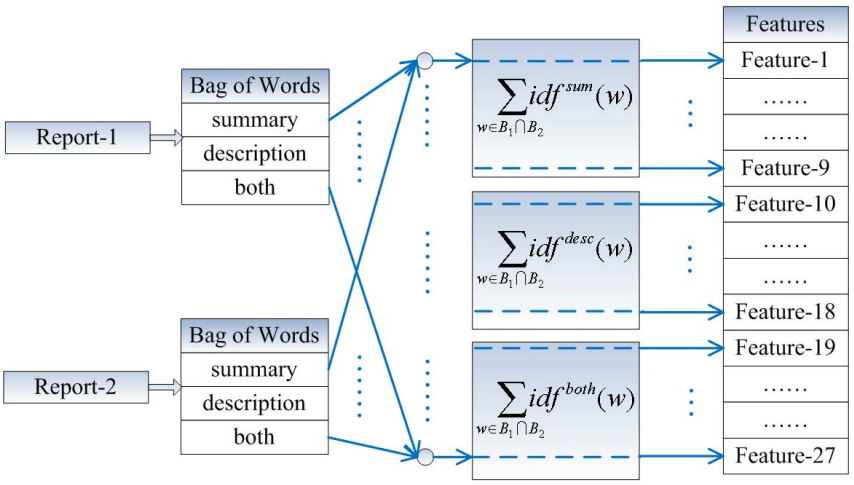


Figure 5: Feature Extraction: First 27 Features

report. In general, it iterates over all the buckets in the repository and calculates the similarity between the new report and each bucket. At last, a list of masters whose buckets have the biggest similarity is returned. In Line 2, $Buckets(Rep)$ returns all the buckets in the bug repository, and $Master(B)$ in Line 4 is the master report of bucket B. The algorithm makes a call to Algorithm 2 which computes the similarity between a report and a bucket of reports. To make the algorithm efficient, we only keep top N candidate buckets in memory while the buckets in the repository are being analyzed. This is achieved by a minimum heap of maximum size N. If the size of $Candidates$ is less than N, new bucket will be directly added. When the size equals to N, if the new bucket whose similarity is greater than the minimum similarity in $Candidates$, it will replace the bucket with the minimum similarity; otherwise, the request of adding the bucket will be ignored by $Candidates$.

Given a new report Q and a bucket B, Algorithm 2 returns the similarity between Q and B. This algorithm first creates candidate duplicate pairs between Q and all the reports in the bucket B (Line 2). Each pair is represented by a vector of features, which is calculated by the function $f54$. The trained discriminative model is used to predict the probability for each candidate pair. Finally, the maximum probability between Q and the bug reports in B is returned as the similarity between Q and B. $Reports(B)$ denotes all the reports in B. The operation $f54(Q, R)$ returns a vector of the 54 similarity features from the pair of bug reports Q and R, including 27 features based on single words and 27 features based on bigrams described in Sub-section 3.3.2. The procedure $SVMPredict$ in Line 4 is the invocation to the discriminative model and it returns a probability in which the pair of reports the feature vector t corresponds to are duplicates of each other.

3.5 Model Evolution

As time passes, new bug reports will come and be triaged. This new information could be used as new training data to update the model. Users could perform this process periodically or every time after a new bug report has been triaged. In general, such newly created training data should be useful. However, we find that such information is not always beneficial to us for the following reason: our retrieval model is based on lexical similarity rather than semantic similarity of text. In another word, if two bug reports refer to the same defect but use different lexical representations, i.e. words, then our retrieval model does not give a high similarity measure to this pair of bug reports. Therefore, including a pair of duplicate bug reports that are not lexically similar in the training data would actually bring noise to our trained classifier. We therefore consider the following two kinds of update:

1. Light update. If our retrieval engine fails to retrieve the right master for a new report before the triager marks the report as a duplicate, we perform this update. When the failure happens, the new bug report could syntactically be very far from the master. For this case, we only perform a light update to the model by updating the *idf* scores of the training examples and re-training the model.

2. Regular update. We perform this update if our retrieval engine is able to retrieve the right master for the new duplicate bug report. When this happens, the new bug report is used to update the *idf* AND to create new training examples. An updated discriminative model is then trained based on the new *idf* and training examples.

Via our experiments, we have empirically validated that the above heuristics work effectively in improving the quality of the model after updating.

4. CASE STUDIES

We apply our discriminative model approach to three large bug repositories of open source projects, OpenOffice, Firefox and Eclipse. For comparison purpose, we also implemented the algorithms in [17, 22, 11] to the best of our knowledge. To evaluate the performance of different approaches, we employed the notion of *recall rate* defined in [17].

$$recall\ rate = \frac{N_{detected}}{N_{total}} \quad (4)$$

(4) shows how *recall rate* is calculated. $N_{detected}$ is the number of duplicate reports whose masters are successfully

detected, while N_{total} is the total number of duplicate reports for testing the retrieval process. Given a candidate list size, the *recall rate* can be interpreted as the percentage of duplicates whose masters are successfully retrieved in the list. In terms of this measure, the result shows that our approach can bring remarkable improvement over the other three approaches.

4.1 Experimental Setup

In our experiment, we used the bug repositories of three large open source projects: the Eclipse project, the Firefox project and the OpenOffice project. Among the three projects, Eclipse is a popular open source integrated development environment written in Java; Firefox is a well-known open source web browser written in C/C++, and OpenOffice is an open source counterpart of Microsoft Office.

These three projects are from different domains, written in different languages and used by different types of users. Thus, carrying out the experiment on them helps to generalize our conclusions. On top of that, all of the three projects have large bug repositories so as to provide ample data for an evaluation. We selected a subset of each repository including both defect reports and feature requests within a period of time to set up an experimental bug set in our study. Specifically, we use the bug report set submitted to OpenOffice in year 2008 (including 12,732 bug reports), the bug report set of Eclipse in year 2008 (including 44,652 bug reports) to evaluate our approach. Furthermore, to study how our training based approach works in the long run (in case the property of bug reports change over time), we further evaluated our approach on the whole bug report set of Firefox (including 47,704 bug reports submitted since Firefox was started in 2002) before June 2007 as a long-run evaluation set.

Table 2 gives the details of the three datasets. We refer to the time period of the datasets as *Time Frame*, as the second column *Time Frame* displays. To run our approach, we select the first M reports of which 100 reports are duplicates as training set to train a discriminative model. The third column of *Training Reports* in the table shows the ratio of duplicates and all reports in the training set. Besides of serving as a training set in our approach, those M reports are also used to simulate the initial bug repository for all experiment runs.

Selecting reports within a time frame introduces a problem. If a duplicate report r_1 is within the time frame while its master is not, then r_1 is not detectable as its master is not in the dataset, which will only decrease the recall rate. For this case, we simply re-mark r_1 as non-duplicate. However, there is a more complex case. Suppose there are two or more duplicate reports on the same defect within the frame while their common master is excluded from the dataset. If we still simply mark them as non-duplicate, we will lose two or more duplicates. So in this case, we first make the oldest report r_{oldest} become the master report, and then mark the others as duplicates of r_{oldest}.

4.2 Experimental Details and Result

We evaluate the performance of our approach as compared to previous techniques over the three datasets. The previous approaches come with a parameter setting corresponding to the weight being assigned to the words from the summary and those from the description of the bug reports. They consider two weights, one is: equal weight between the summary and description, second is: summary carries double weight.

Also, in the three approaches, they consider a non bucket-based retrieval, where relevant bug reports are retrieved rather than relevant buckets. As one bucket contains bug reports corresponding to the same defect, we believe it is best to consider bucket-based retrieval. Report-based retrieval could potentially return more than one report referring to the same defect in the list of candidate duplicate bug reports causing redundant effort for the triager. Also, in [22], the authors use both IR and execution trace to detect duplicate reports. As execution trace is hard to get for existing reports and especially for our large datasets, we only compare based on textual summary and description of the bug reports.

In the three datasets, training reports are used as initial bug repository. They are also used to construct the training set and further to build a discriminative model. At each experimental run, we iterate over the testing reports in chronological order. Once reaching a duplicate report R, we apply the corresponding technique to get a top N list of R's potential master reports. After each detection is done, we record the result whether R's master is detected successfully for future *recall rate* calculation, and then add R to the repository. After the last iteration is over, the *recall rates* for different top list size are calculated.

Figure 6 shows the experiment results of the seven runs on the three datasets. In the figure, the horizontal axis is the top N list's size, and the vertical axis is the recall rate; J stands for [11], R stands for [17] and W is [22]. Suffix -1 or -2 represents weighing summaries 1 or 2. O corresponds to our approach. From the three sub figures, we can easily come to the first conclusion that our technique brings evidence of improvement. We have 17–31% relative improvement in OpenOffice dataset, 22–26% in Firefox dataset and 35–43% in Eclipse dataset. The second conclusion is that manually empirically weighing summaries for traditional IR techniques can help improve limited performance as the six curves near the bottom of each sub figure are very close to one another.

Compared to other six runs, the improvement achieved in our run is due to (1) the 54 similarity features to comprehensively measure how similar two reports are (2) and the use of discriminative model to automatically assign weights to each feature to discriminate duplicate reports from non-duplicate ones, which is rigorous and has theoretical support from machine learning area.

5. DISCUSSION

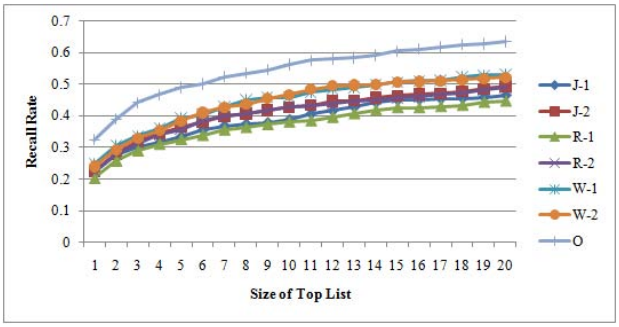
This section will first discuss issues related to runtime overhead, followed by the rationale behind the choice of the 54 similarity scores as features rather than other types of similarity scores.

5.1 Runtime Overhead

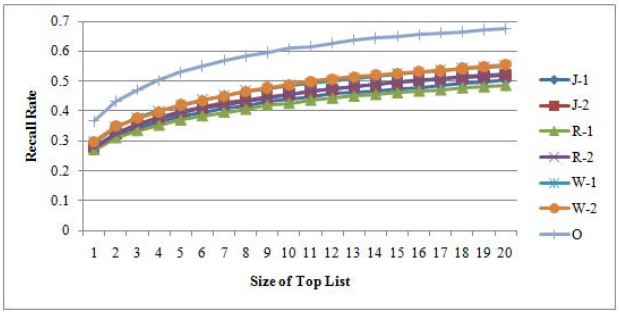
There is no free lunch. With the big increase of recall rates, the runtime overhead of our detection algorithm is also higher than past approaches. In the largest dataset Firefox in the experiment run of our approach, the initial cost to detect a duplicate report is one second. However, as the training set continually grows and the repository gets increasingly large, the detection cost also increases over time. When the experiment considers detecting the last duplicate

Table 2: Summary of Datasets

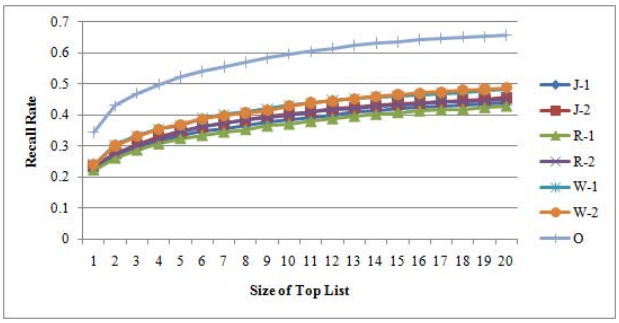
Dataset	Time Frame	Training Reports (Duplicate/All)	Testing Reports(Duplicate/All)
OpenOffice	Jan/02/2008–Dec/30/2008	100/3160	529/9572
Firefox	Apr/04/2002–Jul/07/2007	100/962	3207/46359
Eclipse	Jan/02/2008–Dec/30/2008	100/4265	1913/40387



(a) OpenOffice



(b) FireFox



(c) Eclipse

Figure 6: Recall Rates Comparison between Various Techniques with Certain Top List Size

bug report, the cost becomes *one* minute.

The major overhead is due to the fact that we consider 54 different similarity features between reports. In contrast, previous works [17, 22] consider the similarity between summary+description and summary+description, and [11] considers the similarity between summary and summary, and the similarity between description and description. The runtime overhead is higher at the later part of the experiment as the number of bug report pairs in the training set is larger. Consequently, SVM will need more time to build a discriminative model.

However, a higher runtime overhead does not mean that this approach is not practical. In fact, it is acceptable for real world bug triaging process for two reasons. First, we have experimented with real world datasets. The dataset from Firefox spans from April 04, 2002 to July 07, 2007 and contains more than 47,000 reports in total. For an active software project, considering reports in one-year frame is enough as bug reports are usually received for new software releases. Although the Eclipse dataset with 44652 reports is within one-year time frame, it contains multiple sub-projects, which means that it can be split into smaller datasets. Second, new bug reports do not come every minute. In our three datasets, the average frequency of new report arrival for OpenOffice is 1.5 reports/hour, the one for Firefox is 1 report/hour, and the one for Eclipse is 5 reports/hour. Therefore, our system still has enough time to retrain the model before processing a new bug report.

5.2 Feature Selection

Feature selection is the process to identify a set of features which can bring the best classification performance. A commonly used metric for feature selection is Fisher score [16], defined in the following formula,

$$F_r = \frac{\sum_{i=1}^{c} n_i(\mu_i - \mu)^2}{\sum_{i=1}^{c} n_i \sigma_i^2} \quad (5)$$

where n_i is the number of data examples in class i, μ_i is the average feature value in class i, σ_i is the standard deviation of the feature value in class i, and μ is the average feature value in the whole dataset. Assume x_{ij} is the attribute value for the jth instance in class i, then μ, μ_i and σ_i are defined as $\mu = \frac{\sum_i \sum_j x_{ij}}{\sum_i n_i}$, $\mu_i = \frac{\sum_j x_{ij}}{n_i}$, $\sigma_i = \sqrt{\frac{\sum_j (x_{ij} - \mu_i)^2}{n_i}}$, respectively. The higher the fisher score the better is the feature for classification.

In our approach, we use 54 *idf*-based formulas to calculate similarity features between two bug reports. One might ask why tf or $tf * idf$ measures are not used in the formulas. The reason to choose an *idf*-only solution is that this setting yields the best performance during our empirical validation. To further demonstrate that *idf* is a good measure, we replace the *idf* measure in the 54 features with tf and $tf * idf$ measure respectively. We then calculate the Fisher score of each of the 54 features using *idf*, tf, and $tf * idf$. The results on Eclipse dataset shows that *idf*-based formulas outperform tf-based and $(tf * idf)$-based formulas in terms of Fisher score. This supports our decision in choosing the *idf*-based formulas as feature scores.

6. RELATED WORK

One of the pioneer studies on duplicate bug report detection is by Runeson et al. [17]. Their approach first cleaned the textual bug reports via natural language processing techniques – tokenization, stemming and stop word removal.

The remaining words were then modeled as a vector space, where each axis corresponded to a unique word. Each report was represented by a vector in the space. The value of each element in the vector was computed by the following formula on the tf value of the corresponding word.

$$weight(word) = 1 + log_2(tf(word))$$

After these vectors were formed, they used three measures – cosine, dice and jaccard – to calculate the distance between two vectors as the similarity of the two corresponding reports. Given a bug report under investigation, their system would return top-k similar bug reports based on the similarities between the new report and the existing reports in the repository. A case study was performed on defect reports at Sony Ericsson Mobile Communications, which showed that the tool was able to identify 40% of duplicate bug reports. It also showed that cosine outperformed the other two similarity measures.

In [22], Wang et al. extended the work by Runeson et al. in two dimensions. First they considered not only TF, but IDF. Hence, in their work, the value of each element in a vector corresponded to the following formula,

$$weight(word) = tf(word) * idf(word)$$

Second, they considered execution information to detect duplicates. A case study based on cosine similarity measure on a small subset of bug reports from Firefox showed that their appraoch could detect 67%–93% of duplicate bug reports by utilizing both natural language information and execution traces.

In [11], Jalbert and Weimer extended the work by Runeson et al. in two dimensions. First, they proposed a new term-weighting scheme for the vector representation,

$$weight(word) = 3 + 2 * log_2(tf(word))$$

The cosine similarity was adopted to extract top-k similar reports. Aside from the above, they also adopted clustering and classification techniques to filter duplicates.

Similarities and Differences. Similar to the above three studies, we address the problem of retrieving similar bug reports from repositories for duplicate bug report identification. Similar to the work in [17, 11], we only consider natural language information which is widely available. The execution information considered in [22] is often not available and hard to collect, especially for binary programs. For example, in the OpenOffice, Firefox and Eclipse datasets used in our experiment, the percentages of reports having execution information are indeed low (0.82%, 0.53% and 12% respectively). Also, for large bug repositories, creation of execution information for legacy reports can be time consuming.

Compared to the three approaches, there are generally three differences. First, to retrieve top-k similar repots, they used similarity measure to compute distances between reports while we adopt an approach based on discriminative model. We train a discriminative model via SVMs to classify whether two bug reports are duplicates of one other with a probability. Based on this probability score, we retrieve and rank candidate duplicate bug reports. What is more, The approach in [11] also employed a classifier trained by SVMs, but for a different purpose. Their classifier was used to return a boolean flag of DUPLICATE or NEW for a new report, if the flag was DUPLICATE, the new report would be filtered and would not reach triagers. They reported that 8% of the duplicate reports could be filtered in their experiment. The second difference is that we introduce 54 features, out of which 27 are based on bigrams, to better discriminate duplicate bug reports. Then the trained discriminative model can automatically infer optimum weights for each feature. This mechanism enables our approach to be robust to bad features, adaptive to report repository evolution over time, and effective for different software projects. Finally, instead of returning similar bug reports, we return similar buckets. As each bucket represents a distinct defect, our approach can easily avoid two or more reports in the top-k similar report list referring to the same defect.

From the point of performance view, we show that our discriminative model approach outperforms all previous approaches using natural language information alone by up to 43% on bug report repositories of three large open source applications including Firefox, Eclipse, and OpenOffice.

Besides the effort on duplicate bug report detection, there has also been effort on bug report mining. Anvik et al. [1] and Cubranic and Murphy [8] and Lucca [9] all proposed semi-automatic techniques to categorize bug reports. Based on categories of bug reports, their approaches helped assign bug reports to suitable developers. Menzies and Marcus also suggested a classification based approach to predicting the severity of bug reports [14]. Bettenburg et al. proposed a work on extracting structural information including stack traces, patches, and codes from the descriptions of the bug reports [5]. Ko and Myers analyzed the linguistic characteristics of bug-report summaries, and proposed a technique to differentiate between failure reports and feature calls [12]. They also emphasized on the need of duplicate bug report detection, but did not propose a solution. While the above works mentioned the need of duplicate bug detection or some of their techniques may benefit duplicate bug detection, none of them worked directly on the duplicate bug detection problem.

There have been several statistical studies and surveys of existing bug repositories. Anvik et al. reported a statistical study on open bug repositories with some interesting results such as the proportion of different resolutions and the number of bug reports that a single reporter submitted [2]. Sandusky et al. studied the relationships between bug reports and reported some statistic results on duplicate bugs in open bug repositories [18]. Additionally, Hooimeijer and Weimer suggested a statistics based model to predict the quality of bug reports [10]. After that, Bettenburg et al. made a survey on the developers of several well-known open source projects (Eclipse, Mozilla, and Apache) to study the factors that developers cared most on dealing with bug reports [3]. Bettenburg et al. also suggested that duplicate bug reports were actually not harmful but useful for the developers [4]. So the requirement for duplicate bug report detection became even stronger because it could not only reduce the waste of developer's time on duplicate bug reports but also helped developers to gather more related information to solve the bug more quickly. In general, none of these work proposed any approaches to duplicate-bug-report detection, but some of the work pointed out the motivation and effect of detecting duplicate bug reports.

7. CONCLUSION & FUTURE WORK

In this work, we consider a new approach to detecting

duplicate bug reports by building a discriminative model that answers the question "Are two bug reports duplicates of each other?". The model would report a score on the probability of A and B being duplicates. This score is then used to retrieve similar bug reports from a bug report repository for user inspection. We have investigated the utility of our approach on 3 sizable bug repositories from 3 large open-source applications including OpenOffice, Firefox, and Eclipse. The experiment shows that our approach outperforms existing state-of-the-art techniques by a relative improvement of 17–31%, 22–26%, and 35–43% on OpenOffice, Firefox, and Eclipse dataset respectively.

As a future work, we plan to investigate the utility of paraphrases in discriminative models for potential improvement in accuracy. We have developed a technique to extract technical paraphrases [21] and is currently investigating their utility in improving detection of duplicate bug reports. What is more, an interesting direction is incorporating response threads to bug reports as further sources of information. The other interesting direction is adopting pattern-based classification [7, 13] to extract richer feature set that enables better discrimination and detection of duplicate bug reports.

8. REFERENCES

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

[2] 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*, pages 35–39, 2005.

[3] 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*, pages 308–318, 2008.

[4] 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*, pages 337–345, 2008.

[5] 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*, pages 27–30, 2008.

[6] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

[7] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *ICDE*, 2007.

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

[9] L. G. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 93, 2002.

[10] 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*, pages 34–43, 2007.

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

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

[13] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *proceedings of the SIGKDD Conference on Knowledge Discovery and Data Mining*, 2009.

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

[15] R. Nallapati. Discriminative models for information retrieval. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, 2004.

[16] P. R.Duda and D.Stork. *Pattern Classification*. Wiley Interscience, 2nd edition, 2000.

[17] 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.

[18] 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.

[19] J. Sutherland. Business objects in corporate information systems. In *ACM Computing Surveys*, 2006.

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

[21] X. Wang, D. Lo, J. Jing, L. Zhang, and H. Mei. Extracting Paraphrases of Technical Terms from Noisy Parallel Software Corpora. In *proceedings of the Joint conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, 2009.

[22] 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.