9-2014

DupFinder: Integrated tool support for duplicate bug report detection

Ferdian THUNG
Singapore Management University, ferdiant.2013@smu.edu.sg

Pavneet Singh KOCHHAR
Singapore Management University, kochharps.2012@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

DupFinder: Integrated Tool Support for Duplicate Bug Report Detection

Ferdian Thung, Pavneet Singh Kochhar, and David Lo
School of Information Systems
Singapore Management University
{ferdiant.2013,kochharps.2012,davidlo}@smu.edu.sg

ABSTRACT

To track bugs that appear in a software, developers often make use of a bug tracking system. Users can report bugs that they encounter in such a system. Bug reporting is inherently an uncoordinated distributed process though and thus when a user submits a new bug report, there might be cases when another bug report describing exactly the same problem is already present in the system. Such bug reports are duplicate of each other and these duplicate bug reports need to be identified. A number of past studies have proposed a number of automated approaches to detect duplicate bug reports. However, these approaches are not integrated to existing bug tracking systems. In this paper, we propose a tool named DupFinder, which implements the state-of-the-art unsupervised duplicate bug report approach by Runeson et al., as a Bugzilla extension. DupFinder does not require any training data and thus can easily be deployed to any project. DupFinder extracts texts from summary and description fields of a new bug report and recent bug reports present in a bug tracking system, uses vector space model to measure similarity of bug reports, and provides developers with a list of potential duplicate bug reports based on the similarity of these reports with the new bug report. We have released DupFinder as an open source tool in GitHub, which is available at: `https://github.com/smagsmu/dupfinder`.

Categories and Subject Descriptors: D.2.7 [**Software**]: Software Engineering – *Distribution, Maintenance, and Enhancement*

General Terms: Management; Reliability

Keywords: Bugzilla; Duplicate bug reports; Integrated tool support

1. INTRODUCTION

Bug tracking systems like Bugzilla are used by a large number of developers and organisations to track bugs related to their projects. Well-known projects contain large number of bug reports as they have a big user base, who report issues they encounter while using the software. When a user logs a bug report, there might be cases when another bug report describing the same problem has already been reported earlier by other users. In such cases, this new bug report is a duplicate bug report. Usually, projects have bug triagers who check for duplicate bug reports. The number of bug reports reported daily however can be too large for triagers to handle. Anvik et al. quoted a Mozilla developer who commented that "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle". This highlights the need for an automated tool support to help detect and prevent duplicate bug reports.

Researchers have proposed several techniques to address the issue of duplicate bug reports. These techniques can be categorized into two families: unsupervised and supervised. Runeson et al. propose an unsupervised technique that takes a new bug report and returns a ranked list of top-k most similar reports to help detect duplicate reports [8]. Being an unsupervised technique, Runeson et al.'s approach does not require any training data and thus can be used for any bug tracking systems even those with a small number of bug reports. There are also a number of supervised techniques proposed in the literature which require a set of training data (i.e., sets of bug reports that are known to be duplicate of one another). For example, Jalbert and Weimer propose a technique to automatically classify bug reports as duplicate or not using a classifier learned from a training data [2]. Also, Sun et al. propose a retrieval function which uses textual content from summary and description and non-textual fields such as product, component etc. to measure the similarity between bug reports [9]. Sun et al.'s approach eventually produces a ranked list of bug reports that are deemed to be most similar to a new bug report. It makes use of a training data to tune the weights of the various fields to achieve a higher accuracy.

Each of the above studies have been evaluated on a large number of bug reports and shown to be effective. However, since their implementations are only proof of concepts, the implementations are often minimal and have not been designed with actual deployment picture in mind. None of the implementations are actually integrated to an existing bug tracking system. Thus, practitioners cannot download and use these techniques to deal with the problem of duplicate bug reports that they encounter in their day-to-day development practice.

To address the above problem, we develop a tool DupFinder, which is implemented as a Bugzilla extension,

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2648627.

to search for duplicate bug reports. Our goal is not to design a new algorithm but rather to implement an existing technique into a tool integrated to a bug tracking system that can be used by practitioners to help them deal with duplicate bug report problem. Our tool is based on the unsupervised technique proposed by Runeson et al. [8]. It uses a vector space model to represent a bug report. It then compares the vector space model representation of a new bug report with existing bug reports to get a list of similar bug reports. As a user enters text in the summary or description fields of a new bug report, our tool calculates similarity between the entered text and existing bug reports and returns the top bug reports that are most similar to the text.

The structure of the remainder of this paper is as follows. In Section 2, we briefly describe Bugzilla and vector space model. In Section 3, we describe our tool DupFinder. Related work is presented in Section 4. Section 5 concludes and describes future work.

2. PRELIMINARIES

In this section, we first briefly describe Bugzilla, a bug tracking system on which our tool is build upon. Next, we describe vector space model (VSM), the model which we use to calculate similarity between bug reports.

2.1 Bugzilla

Bugzilla[1] is a web-based bug tracker. Users can report new bugs and developers can track the status of bugs that are reported. It is used by well-known organisations such as Mozilla and Apache and open-source projects such as Eclipse and LibreOffice.

A bug report in Bugzilla contains a number of fields such as bug id, product, component, assignee, summary and description. Each of them carries a piece of information. In this work, we are interested in the following two textual fields: (1) summary, (2) description. Summary is a short synopsis of a bug, while description is a longer text that elaborates the bug.

2.2 Vector Space Model

To apply vector space model, a textual document is first pre-processed. The pre-processed text is then converted into a representation in the form of a vector of weights. The similarity of two documents can be computed by comparing their representative vectors.

Text Preprocessing: Text preprocessing is an important task in information retrieval. There are three common text preprocessing steps: tokenization, stop word removal, and stemming. At the end of these steps, each bug report is represented as a bag (i.e., multi-set) of words.

In the tokenization step, we remove special symbols, punctuation marks, and number literals from bug reports. In the stop word removal step, we remove the commonly occurring English words such as "I", "you", "we',' etc. from the bug reports as most of them carry little meaning. We use the stop word list from Lingua::StopWords module in Comprehensive Perl Archive Network (CPAN)[2]. In the stemming step, we reduce all words to their root form. For example, we reduce "mapping", "mapped", and "maps" to "map". To do this, we apply the well known Porter stemming algorithm and use Lingua::Stem module implementation from CPAN.

Representation and Similarity Vector Space Model represents a document as a vector of weights, where each weight corresponds to a word in the document. The weight of each word is usually computed using the product of its term frequency and its inverse document frequency, following the standard tf-idf weighting scheme [6]. The following is the tf-idf weight of word w in document d given a set of documents D (denoted as *tf-idf*(w, d, D)):

$$\textit{tf-idf}(w, d, D) = f(w, d) \times \log \frac{\mid D \mid}{\mid \{d_i | w \in d_i \wedge d_i \in D\} \mid}$$

In the above equation, $f(w, d)$ is the number of times word w occurs in document d, and $w \in d_i$ denotes that word w appears in document d_i. Textual similarity between a document q and another document d is obtained by computing the *cosine similarity* of the two vectors representing q and d [6].

3. DUPFINDER

In this section, we describe the architecture of DupFinder, the algorithm that we use to return potential duplicate bug reports, some implementation details, and a usage scenario on how DupFinder can be used.

System Architecture. DupFinder consists of a client-side component and a server-side component. The client-side component handles interaction with user (i.e., reporter) and communicates with the server-side component to retrieve relevant bug reports given a user query. A user query is a combination of the texts that appear in the summary and the description fields of a new bug report. The client-side component is added to the bug report entry page (i.e., the page where a user can enter the details of a bug that the user intends to report). The server-side component compares the user query with the existing bug reports and outputs a list of most similar reports.

Algorithm. Algorithm 1 shows the pseudocode of an algorithm for finding duplicate bug reports, which is implemented in the server-side component. It accepts as input a new bug report $NewBr$, k most recently created bug reports $RecentReports$, and number of most similar bug reports to return n. The algorithm returns a ranked list of n bug reports in $RecentReports$ that are most similar to $NewBr$. In lines 1-4, it concatenates the text in the summary and description fields of $NewBr$, performs text preprocessing on the concatenated text, and creates a vector space model (VSM) representation (which is a vector of weights) from the preprocessed text. The detail of each of these steps is described in Section 2.2. In lines 5-9, it also performs text concatenation, text preprocessing, and constructs a VSM representation from the summary and description fields of each report in $RecentReports$. In line 10, it computes the cosine similarity between the VSM representation of the new bug report and the VSM representation of each bug report in $RecentReports$. In line 12, it then sorts bug reports in $RecentReports$ based on their cosine similarity scores. Finally, in line 13, top-n bug reports with the highest scores are returned. By default, we set k and n to 100 and 5, respectively. Users might set k to a larger number to reduce the risk of not identifying duplicates of older bugs. Setting

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

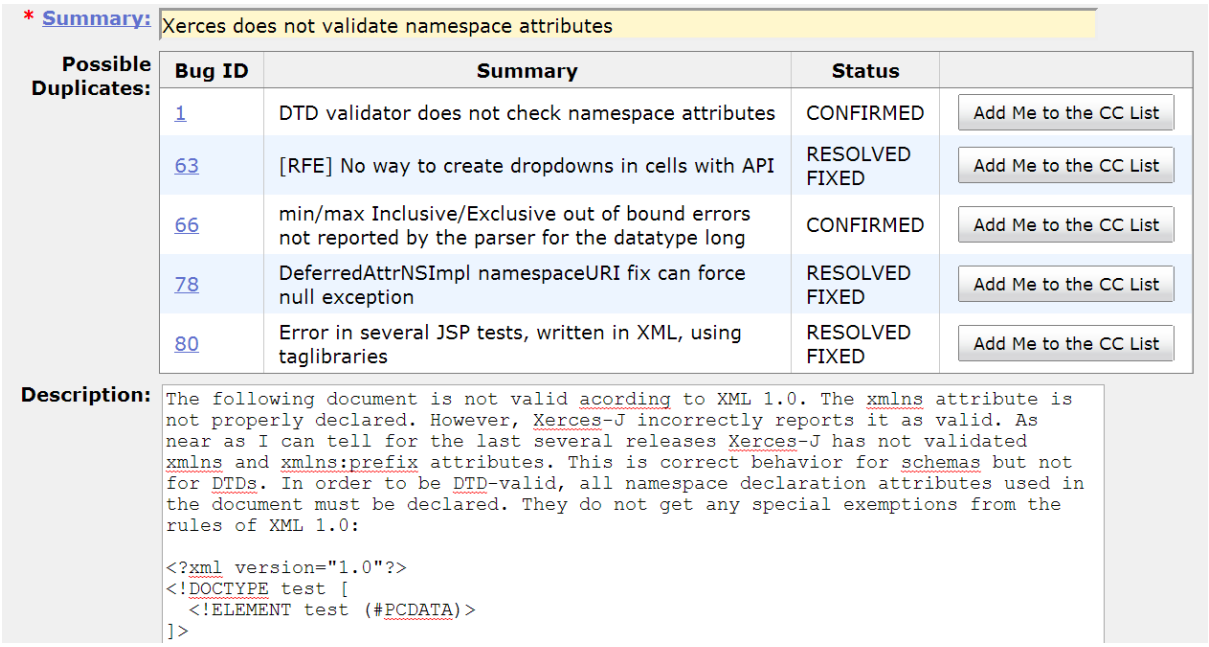

Figure 1: DupFinder User Interface

Algorithm 1 FindDuplicate

Input: $NewBr$: a new bug report
$RecentReports$: k most recent bug reports
n: number of most similar bug reports to return
Output: top-n most similar bug reports

```
1  NewBrTokens ← Tokenize(NewBr.Summary, NewBr.Desc)
2  Remove stop words from NewBrTokens
3  Stem each word in NewBrTokens
4  NewBrVSM ← ConstructVSM(NewBrTokens)
5  foreach instance Br ∈ RecentReports do
6  |  BrTokens ← Tokenize(Br.Summary, Br.Desc)
7  |  Remove stop words from BrTokens
8  |  Stem each word in BrTokens
9  |  BrVSM ← ConstructVSM(BrTokens)
10 |  Br.Sim ← CosineSimilarity(NewBrVSM, BrVSM)
11 end
12 Sort Br in RecentReports by Br.Sim
13 return top-n most similar bug reports
```

k to a larger number increases the runtime cost though. In case the runtime cost is unacceptable for a large k, our tool's efficiency can be improved by building an index.

The algorithm is based on Runeson et al. work [8]. However, we do not implement synonyms and spellchecking part of their approach as it requires domain knowledge and manual labeling (i.e., they survey company employees to construct a thesaurus). We implement only parts of their approach that can be fully automated and thus is applicable to bug reports from a wide-range of software projects.

Implementation Details. The client-side of DupFinder is implemented by overriding a template of Bugzilla that renders the user interface that allows users to input new bug reports. The server-side of DupFinder is implemented as a new web-service that is called by the client-side. These follows the standard procedure specified by Bugzilla to implement a new extension.

Usage Scenario. Figure 1 shows the interface of bug report entry page when our extension is enabled and a new bug report has been entered. As shown in the figure, a user can enter text into the summary and description fields of the new bug report. A background script monitors the state of the summary and description fields. When a user enters a text to either fields, the background script waits until the user stops typing and then seamlessly extracts the content of the summary and description fields. The extracted content is then sent to the server-side component. The background script also concurrently generates a table below the summary field in the bug report entry page. Initially, this table only notifies the reporter that the system is currently searching for possible duplicate bug reports. The table's content will be changed depending on the response from the server-side component. If the server-side component gives no response or returns an error, the table's content will notify the user of the problem. If the server-side component returns a list of relevant bug reports, the table will be filled with information of those returned bug reports. The table will then contain these elements:

1. **Bug ID.** It is the id of an existing bug report that is a possible duplicate of the new bug report. It is linked to a separate page that displays the bug report which allows the user to thoroughly investigate whether the bug report is really describing the bug that the user is currently reporting.
2. **Summary.** It is the synopsis of an existing bug report that is a possible duplicate. It allows for quick assessment of whether the bug report is indeed a duplicate of the new bug report.
3. **Status.** It indicates the status of a possible duplicate bug report. If a user thinks that the bug report is a duplicate and the status is fixed, then the user does not need to proceed any further.
4. **Add CC Button.** It allows a user to add himself to the cc list of the corresponding bug report that is considered a potential duplicate of the currently entered

bug report. This is only relevant if the possible duplicate bug report is still opened and the user wants to keep track or contribute to the bug fixing process.

4. RELATED WORK

In this section, we describe past studies on duplicate bug reports detection which can be divided into two categories: unsupervised and supervised approaches. Unsupervised approaches do not require any training data (i.e., sets of bug reports that are known to be duplicate of one another). On the other hand, supervised approaches require training data.

Unsupervised Approaches. Runeson et al. use an information retrieval technique to identify duplicate bug reports [8]. Their approach uses vector space model to represent a bug report and compares representations of bug reports to identify bug reports that are similar to a new bug report. They develop a prototype tool and evaluate it on a corpus of defect reports from Sony Ericsson Mobile Communications. Sureka and Jalote propose a method which uses N-gram based model to detect duplicate bug reports and evaluate their technique on bug reports from Eclipse [11]. The work of Runeson et al. and Sureka et al. have never been compared against a common dataset and thus it is unclear which technique is the best performing one. Wang et al. extend the work by Runeson et al. to consider execution traces and show that by considering execution traces, duplicate bug report can be identified more accurately [13]. However, execution traces are often unavailable for many bug reports. In this work, we integrate the approach by Runeson et al. into Bugzilla bug tracking system and make our implementation available for public use.

Supervised Approaches. A number of supervised approaches work in the same setting as the above mentioned unsupervised approaches, that is given a new bug report, return a ranked list of bug reports that are most similar to it. Sun et al. build a discriminative model, which gives a score based on the probability of two reports being duplicate of each other and the score is used to retrieve similar bug reports [10]. Sun et al. extend their previous work by proposing a new information retrieval model based on BM25 whose weights are tuned based on a training data [9]. Nguyen et al. extend the work of Sun et al. by combining BM25 with a specialized topic model [7]. Other supervised approaches build a model based on a training data and use it to analyze a new bug report and predict if it is a duplicate bug report or not. Jalbert and Weimer predict duplicate bug reports using text similarity, surface, and clustering features [2]. Tian et al. extend this work by using more effective features (e.g., REP similarity measure, product, etc.) to achieve higher accuracy [12]. Another set of supervised approaches build a model based on a training data and use it to analyze a pair of bug reports and predict whether they are duplicate of each other or not. Lo et al. mine closed discriminative dyadic sequential patterns and use them to detect duplicate bug report pairs [5]. Alipour make use textual, categorical, and contextual features to detect duplicate bug report pairs [1]. Klein et al. and Lazar et al. extend Alipour et al. work by adding new features [3, 4].

Bugzilla Default Functionality. Bugzilla provides a default functionality that also recommends similar reports when a user enters a new bug report. However, the default implementation is only based on text in the summary field of bug reports and does not consider the frequency of words that appear in bug reports. Thus, our tool would rank the bug reports better if: (1) relevant texts are entered only in the description field of the new bug report, or (2) both duplicate and non-duplicate reports of the new bug report contain relevant words, but with different frequencies. Our tool replaces this default implementation with the technique proposed by Runeson et al. which uses state-of-the-art information retrieval techniques. Our goal is to make recent advances in the area of duplicate bug report detection accessible to practitioners.

5. CONCLUSION AND FUTURE WORK

Duplicate bug report detection has been actively researched for many years and thus many techniques have been proposed to solve it. However, they are currently not accessible to practitioners. In this work, we have developed a tool called DupFinder that implements a state-of-the-art unsupervised duplicate bug report detection technique proposed by Runeson et al.. For good integration with an existing bug tracking system, we implement it as a Bugzilla extension. This allows DupFinder to be easily used in actual development environment. In the future, we plan to incorporate more state-of-the-art duplicate bug report detection algorithms that have been proposed in literature. We plan to add both unsupervised and supervised duplicate bug report detection approaches into DupFinder.

6. REFERENCES

[1] A. Alipour, A. Hindle, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *MSR*, 2013.

[2] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *DSN*, 2008.

[3] N. Klein, C. S. Corley, and N. A. Kraft. New features for duplicate bug detection. In *MSR*, 2014.

[4] A. Lazar, S. Ritchey, and B. Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *MSR*, 2014.

[5] D. Lo, H. Cheng, and Lucia. Mining closed discriminative dyadic sequential patterns. In *EDBT*, 2011.

[6] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge, 2008.

[7] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *ASE*, 2012.

[8] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, 2007.

[9] C. Sun, D. Lo, S. C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, 2011.

[10] 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.

[11] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *APSEC*, 2010.

[12] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *CSMR*, 2012.

[13] 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 *ICSE*, 2008.