

# Automated Duplicate Detection for Bug Tracking Systems

Nicholas Jalbert  
University of Virginia  
Charlottesville, Virginia 22904  
jalbert@virginia.edu

Westley Weimer  
University of Virginia  
Charlottesville, Virginia 22904  
weimer@cs.virginia.edu

## Abstract

*Bug tracking systems are important tools that guide the maintenance activities of software developers. The utility of these systems is hampered by an excessive number of duplicate bug reports—in some projects as many as a quarter of all reports are duplicates. Developers must manually identify duplicate bug reports, but this identification process is time-consuming and exacerbates the already high cost of software maintenance. We propose a system that automatically classifies duplicate bug reports as they arrive to save developer time. This system uses surface features, textual semantics, and graph clustering to predict duplicate status. Using a dataset of 29,000 bug reports from the Mozilla project, we perform experiments that include a simulation of a real-time bug reporting environment. Our system is able to reduce development cost by filtering out 8% of duplicate bug reports while allowing at least one report for each real defect to reach developers.*

## 1. Introduction

As software projects become increasingly large and complex, it becomes more difficult to properly verify code before shipping. Maintenance activities [12] account for over two thirds of the life cycle cost of a software system [3], summing up to a total of \$70 billion per year in the United States [16]. Software maintenance is critical to software dependability, and defect reporting is critical to modern software maintenance.

Many software projects rely on bug reports to direct corrective maintenance activity [1]. In open source software projects, bug reports are often submitted by software users or developers and collected in a database by one of several bug tracking tools. Allowing users to report and potentially help fix bugs is assumed to improve software quality overall [13]. Bug tracking systems allow users to report, describe, track, classify and comment on bug reports and feature requests. *Bugzilla* is a particularly popular open

source bug tracking software system [14] that is used by large projects such as Mozilla and Eclipse. Bugzilla bug reports come with a number of pre-defined fields, including categorical information such as the relevant product, version, operating system and self-reported incident severity, as well as free-form text fields such as defect title and description. In addition, users and developers can leave comments and submit attachments, such as patches or screenshots.

The number of defect reports typically exceeds the resources available to address them. Mature software projects are forced to ship with both known and unknown bugs; they lack the development resources to deal with every defect. For example, in 2005, one Mozilla developer claimed that, “everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle” [2, p. 363].

A significant fraction of submitted bug reports are spurious duplicates that describe already-reported defects. Previous studies report that as many as 36% of bug reports were duplicates or otherwise invalid [2]. Of the 29,000 bug reports used in the experiments in this paper, 25.9% were identified as duplicates by the project developers.

Developer time and effort are consumed by the triage work required to evaluate bug reports [14], and the time spent fixing bugs has been reported as a useful software quality metric [8]. Modern software engineering for large projects includes bug report triage and duplicate identification as a major component.

We propose a technique to reduce bug report triage cost by *detecting duplicate bug reports as they are reported*. We build a classifier for incoming bug reports that combines the surface features of the report [6], textual similarity metrics [15], and graph clustering algorithms [10] to identify duplicates. We attempt to predict whether manual triage efforts would eventually resolve the defect report as a duplicate or not. This prediction can serve as a filter between developers and arriving defect reports: a report predicted to be a duplicate is filed, for future reference, with the bug reports it is likely to be a duplicate of, but is not otherwise

presented to developers. As a result, no direct triage effort is spent on it. Our classifier is based on a model that takes into account easily-gathered surface features of a report as well as historical context information about previous reports.

In our experiments we apply our technique to over 29,000 bug reports from the Mozilla project and experimentally validate its predictive power. We measure our approach's efficacy as a filter, its ability to locate the likely original for a duplicate bug report, and the relative power of the key features it uses to make decisions.

The Mozilla project already has over 407,000 existing reports, so naive approaches that explicitly compare each incoming bug report to all previous ones will not scale. We train our model on historical information in long (e.g., four-month) batches, periodically regenerating it to ensure it remains accurate.

The main contributions of this paper are:

- A classifier that predicts bug report duplicate status based on an underlying model of surface features and textual similarity. This classifier has reasonable predictive power over our dataset, correctly identifying 8% of the duplicate reports while allowing at least one report for each real defect to reach developers.
- A discussion of the relative predictive power of the features in the model and an explanation of why certain measures of word frequency are not helpful in this domain.

The structure of this paper is as follows. Section 2 presents a motivating example that traces the history of several duplicate bug reports. We compare our approach to others in Section 3. In Section 4, we formalize our model, paying careful attention to textual semantics in Section 4.1 and surface features in Section 4.2. We present our experimental framework and our experimental results in Section 5. We conclude in Section 6.

## 2. Motivating Example

Duplicate bug reports are such a problem in practice that many projects have special guidelines and websites devoted to them. The "Most Frequently Reported Bugs" page of the Mozilla Project's Bugzilla bug tracking system is one such example. This webpage tracks the number of bug reports with known duplicates and displays the most commonly reported bugs. Ten bug equivalence classes have over 100 known duplicates and over 900 other equivalence classes have more than 10 known duplicates each. All of these duplicates had to be identified by hand and represent time developers spent administering the bug report database and performing triage rather than actually addressing defects.

Bug report #340535 is indicative of the problems involved; we will consider it and three of its duplicates.

The body of bug report #340535, submitted on June 6, 2006, includes the text, "when I click OK the updater starts again and tries to do the same thing again and again. It never stops. So I have to kill the task." It was reported with severity "normal" on Windows XP and included a logfile.

Bug report #344134 was submitted on July 10, 2006 and includes the description, "I got a software update of Minefield, but it failed and I got in an endless loop." It was also reported with severity "normal" on Windows XP, but included no screenshots or logfiles. On August 29, 2006 the report was identified as a duplicate of #340535.

Later, on September 17, 2006, bug report #353052 was submitted: "...[Thunderbird] says that the previous update failed to complete, try again get the same message cannot start thunderbird at all, continous loop." This report had a severity of "critical" on Windows XP, and included no attachments. Thirteen hours later, it was marked as a duplicate in same equivalence class as #340535.

A fourth report, #372699, was filed on March 5, 2007. It included the title, "autoupdate runs...and keeps doing this in an infinite loop until you kill thunderbird." It had a severity of "major" on Windows XP and included additional form text. It was marked as a duplicate within four days.

When these four example reports are presented in succession their similarities are evident, but in reality they were separated by up to nine months and over thirty thousand intervening defect reports. All in all, 42 defect reports were submitted describing this same bug. In commercial development processes with a team of bug triagers and software maintainers, it is not reasonable to expect any single developer to have thirty thousand defects from the past nine months memorized for the purposes of rapid triage. Developers must thus be wary, and the cost of checking for duplicates is paid not merely for actual duplicates, but also for every non-duplicate submission; developers must treat every report as a potential duplicate.

However, we can gain insight from this example. While self-reported severity was not indicative, some defect report features such as platform were common to all of the duplicates. More tellingly, however, the duplicate reports often used similar language. For example, each report mentioned above included some form of the word "update", and included "never stops", "endless loop", "continous loop", or "infinite loop", and three had "tries again", "keeps trying", or "keeps doing this".

We propose a formal model for reasoning about duplicate bug reports. The model identifies equivalence classes based predominantly on textual similarity, relegating surface features to a supporting role.

### 3. Related Work

In previous work we presented a model of defect report quality based only on surface features [6]. That model predicted whether a bug would be triaged within a given amount of time. This paper adopts a more semantically-rich model, including textual information and machine learning approaches, and is concerned with detecting duplicates rather than predicting the final status of non-duplicate reports. In addition, our previous work suffered from false positives and would occasionally filter away all reports for a given defect. The technique presented here suffers from no such false positives in practice on a larger dataset.

Anvik et al. present a system that automatically assigns bug reports to an appropriate human developer using text categorization and support vector machines. They claim that their system could aid a human triager by recommending a set of developers for each incoming bug report [2]. Their method correctly suggests appropriate developers with 64% precision for Firefox, although their datasets were smaller than ours (e.g., 10,000 Firefox reports) and their learned model did not generalize well to other projects. They build on previous approaches to automated bug assignment with lower precision levels [4, 5]. Our approach is orthogonal to theirs and both might be gainfully employed together: first our technique filters out potential duplicates, and then the remaining real bug reports are assigned to developers using their technique. Anvik et al. [1] also report preliminary results for duplicate detection using a combination of cosine similarity and top lists; their method requires human intervention and incorrectly filtered out 10% of non-duplicate bugs on their dataset.

Weiß et al. predict the “fixing effort” or person-hours spent addressing a defect [17]. They leverage existing bug databases and historical context information. To make their prediction, they use pre-recorded development cost data from the existing bug report databases. Both of our approaches use textual similarity to find closely related defect reports. Their technique employs the  $k$ -nearest neighbor machine learning algorithm. Their experimental validation involved 600 defect reports for the JBoss project. Our approach is orthogonal to theirs, and a project might employ our technique to weed out spurious duplicates and then employ their technique on the remaining real defect reports to prioritize based on predicted effort.

Kim and Whitehead claim that the time it takes to fix a bug is a useful software quality measure [8]. They measure the time taken to fix bugs in two software projects. We predict whether a bug will eventually be resolved as a duplicate and are not focused on particular resolution times or the total lifetime of real bugs.

Our work is most similar to that of Runeson et al. [15], in which textual similarity is used to analyze known-duplicate

bug reports. In their experiments, bug reports that are known to be duplicates are analyzed along with a set of historical bug reports with the goal of generating a list of candidate originals for that duplicate. In Section 5.2 we show that our technique is no worse than theirs at that task. However, our main focus is on using our model as a filter to detect unknown duplicates, rather than correctly binning known duplicates.

### 4. Modeling Duplicate Defect Reports

Our goal is to develop a model of bug report similarity that uses easy-to-gather surface features and textual semantics to predict if a newly-submitted report is likely to be a duplicate of a previous report. Since many defect reports are duplicates (e.g., 25.9% in our dataset), automating this part of the bug triage process would free up time for developers to focus on other tasks, such as addressing defects and improving software dependability.

Our formal model is the backbone of our bug report filtering system. We extract certain features from each bug report in a bug tracker. When a new bug report arrives, our model uses the values of those features to predict the eventual duplicate status of that new report. Duplicate bugs are not directly presented to developers to save triage costs.

We employ a linear regression over properties of bug reports as the basis for our classifier. Linear regression offers the advantages of (1) having off-the-shelf software support, decreasing the barrier to entry for using our system; (2) supporting rapid classifications, allowing us to add textual semantic information and still perform real-time identification; and (3) easy component examination, allowing for a qualitative analysis of the features in the model. Linear regression produces continuous output values as a function of continuously-valued features; to make a binary classifier we need to specify those features and an output value cutoff that distinguishes between duplicate and non-duplicate status.

We base our features not only on the newly-submitted bug report under consideration, but also on a corpus of previously-submitted bug reports. A key assumption of our technique is that these features will be sufficient to separate duplicates from non-duplicates. In essence, we claim that there are ways to tell duplicate bug reports from non-duplicates just by examining them and the corpus of earlier reports.

We cannot update the context information used by our model after every new bug report; the overhead would be too high. Our linear regression model speeds up processing incoming reports because coefficients can be calculated ahead of time using historic bug data. A new report requires only feature extraction, multiplication, and a test against a cutoff. However, as more and more reports are submitted

the original historic corpus becomes less and less relevant to predicting future duplicates. We thus propose a system in which the basic model is periodically (e.g., yearly) regenerated, recalculating the coefficients and cutoff based on an updated corpus of reports.

We now discuss the derivation of the important features used in our classifier model.

## 4.1 Textual Analysis

Bug reports include free-form textual descriptions and titles, and most duplicate bug reports share many of the same words. Our first step is to define a textual distance metric for use on titles and descriptions. We use this metric as a key component in our identification of duplicates.

We adopt a “bag of words” approach when defining similarity between textual data. Each text is treated as a set of words and their frequency: positional information is not retained. Since orderings are not preserved, some potentially-important semantic information is not available for later use. The benefit gained is that the size of the representation grows at most linearly with the size of the description. This reduces processing load and is thus desirable for a real-time system.

We treat bug report titles and bug report descriptions as separate corpora. We hypothesize that the title and description have different levels of importance when used to classify duplicates. In our experience, bug report titles are written more succinctly than general descriptions and thus are more likely to be similar for duplicate bug reports. We would therefore lose some information if we combined titles and descriptions together and treated them as one corpus. Previous work presents some evidence for this phenomenon: experiments which double the weighting of the title result in better performance [15].

We pre-process raw textual data before analyzing it, tokenizing the text into words and removing stems from those words. We use the MontyLingua tool [9] as well as some basic scripting to obtain tokenized, stemmed word lists of description and title text from raw defect reports. Tokenization strips punctuation, capitalization, numbers, and other non-alphabetic constructs. Stemming removes inflections (e.g., “scrolls” and “scrolling” both reduce to “scroll”). Stemming allows for a more precise comparison between bug reports by creating a more normalized corpus; our experiments used the common Porter stemming algorithm (e.g., [7]).

We then filter each sequence against a stoplist of common words. Stoplists remove words such as “a” and “and” that are present in text but contribute little to its comparative meaning. If such words were allowed to remain, they would artificially inflate the perceived similarity of defect reports with long descriptions. We used an open source stoplist of

roughly 430 words associated with the ReqSimile tool [11].

Finally, we do not consider submission-related information, such as the version of the browser used by the reporter to submit the defect report via a web form, to be part of the description text. Such information is typically colocated with the description in bug databases, but we include only textual information explicitly entered by the reporter.

### 4.1.1 Document Similarity

We are interested in measuring the similarity between two documents within the same corpus; in our experiments all of the descriptions form one corpus and all of the titles form another. All of the documents in a corpus taken together contain a set of  $n$  unique words. We represent each document in that corpus by a vector  $v$  of size  $n$ , with  $v[i]$  related to the total number of times that word  $i$  occurs in that document. The particular value at position  $v[i]$  is obtained from a formula that can involve the number of times word  $i$  appears in that document, the number of times it appears in the corpus, the length of the document, and the size of the corpus.

Once we have obtained the vectors  $v_1$  and  $v_2$  for two documents in the same corpus, we can compute their similarity using the following formula in which  $v_1 \bullet v_2$  represents the dot product:

$$\text{similarity} = \cos(\theta) = \frac{v_1 \bullet v_2}{|v_1| \times |v_2|}$$

That is, the closer two vectors are to colinear, then the more weighted words the corresponding documents share and thus, we assume, the more similar the meanings of the two documents. Given this cosine similarity, the efficacy of our distance metric is determined by how we populate the vectors, and in particular how we weight word frequencies.

### 4.1.2 Weighting for Duplicate Defect Detection

Inverse document frequency, which incorporates corpus-wide information about word counts, is commonly used in natural language processing to identify and appropriately weight important words. It is based on the assumption that important words are distinguished not only by the number of times they appear in a certain text, but also by the inverse of the ratio of the documents in which they appear in the corpus. Thus a word like “the” may appear multiple times in a single document, but will not be heavily-weighted if it also appears in most other documents. The popular TF/IDF weighting includes both normal term frequency within a single document as well as inverse document frequency over an entire corpus.

In Section 5 we present experimental evidence that inverse document frequency is not effective at distinguishing duplicate bug reports. In our dataset, duplicate bug reports











