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

1-4244-2398-9/08/$20.00 ©2008 IEEE

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 naïve 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 duplictes.

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:

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

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 of the same underlying defect are no more likely to share "rare" words than are otherwise-similar unrelated pairs of bug reports. We thus do not include a weighting factor corresponding to inverse document frequency. Our weighting equation for textual similarity is:

$$w_i = 3 + 2\log_2(\text{count of word } i \text{ in document})$$

Every position i in the representative vector of a bug report v is determined based upon the frequency of term i and the constant scaling factors present in the equation. Intuitively, the weight of a word that occurs many times grows logarithmically, rather than linearly. The constant factors were empirically derived in an exhaustive optimization related to our dataset, which ranges over all of the subprojects under the Mozilla umbrella. Once we have each document represented as a weighted vector v, we can use cosine similarity to obtain a distance between two documents.

A true distance metric is symmetric. However, we use a non-symmetric "similarity function" for our training data: textual distance is used as defined above in general, but as a special case the one-directional similarity of an original to its duplicates is set to zero. We hypothesize that duplicates will generally be more similar to the original bug report than to unrelated reports. Because we are predicting if a report is a duplicate and only one part of a duplicate-original pair has that feature, textual similarity would be somewhat less predictive if it were symmetric.

4.1.3 Clustering

We use our textual similarity metric to induce a graph in which the nodes are defect reports and edges link reports with similar text. We then apply a clustering algorithm to this graph to obtain a set of clustered reports. Many common clustering algorithms require either that the number of clusters be known in advance, or that clusters be completely disjoint, or that every element end up in a non-trivial cluster. Instead, we chose to apply a graph clustering algorithm designed for social networks to the problem of detecting duplicate defect reports.

The graph cluster algorithm of Mishra et al. produces a set of possibly-overlapping clusters given a graph with unweighted, undirected edges [10]. Every cluster discovered is internally dense, in that nodes within a cluster have a high fraction of edges to other nodes within the cluster, and also externally sparse, in that nodes within a cluster have a low fraction of edges to nodes not in the cluster. The algorithm is designed with scalability in mind, and has been used to cluster graphs with over 500,000 nodes. We selected it because it does not require foreknowledge of the number of clusters, does not require that every node be in a non-trivial cluster, and is efficient in practice.

In addition, the algorithm produces a "champion", or exemplary node within the cluster that has many neighbors within the cluster and few outside of it. In our experiments in Section 5 we measure our predictive power as a duplicate classifier. In practice our distance metric and the champions of the relevant clusters can also be used to determine which bug from an equivalence class of duplicates should be presented to developers first.

We obtain the required graph by choosing a cutoff value. Nodes with similarity above the cutoff value are connected by an edge. The cutoff similarity and the clustering parameteres used in our experiments were empirically determined.

4.2 Model Features

We use textual similarity and the results of clustering as features for a linear model. We keep description similarity and title similarity separate. For the incoming bug report under consideration, we determine both the highest title similarity and highest description similarity it shares with a report in our historical data. Intuitively, if both of those values are low then the incoming bug report is not textually similar to any known bug report and is therefore unlikely to be a duplicate.

We also use the clusters from Section 4.1.3 to define a feature that notes whether or not a report was included in a cluster. Intuitively, a report left alone as a singleton by the clustering algorithm is less likely to be a duplicate. It is common for a given bug to have multiple duplicates, and we hope to tease out this structure using the graph clustering.

Finally, we complete our model with easily-obtained surface features from the bug report. These features include the self-reported severity, the relevant operating system, and the number of associated patches or screenshots [6]. These features are neither as semantically-rich nor as predictive as textual similarity. Categorical features, such as relevant operating system, were modeled using a one-hot encoding.

5. Experiments

All of our experiments are based on 29,000 bug reports from the Mozilla project. The reports span an eight month period from February 2005 to October 2005. The Mozilla project encompasses many different programs including web browsers, mail clients, calendar applications, and issue tracking systems. All Mozilla subprojects share the same bug tracking system; our dataset includes reports from all of them. We chose to include all of the subprojects to enhance the generality of our results.

Mozilla has been under active development since 2002. Bug reports from the very beginning of a project may not be representative of a "typical" or "steady state" bug report. We selected a time period when a reasonable number of bug reports were resolved while avoiding start-up corner cases. In our dataset, 17% of defect reports had not attained some sort of resolution. Because we use developer resolutions as a ground truth to measure performance, reports without resolutions are not used. Finally, when considering duplicates in this dataset we restrict attention to those for which the original bug is also in the dataset.

We conducted four empirical evaluations:

Text. Our first experiment demonstrates the lack of correlation between sharing "rare" words and duplicate status. In our dataset, two bug reports describing the same bug were no more likely to share "rare" words than were two non-duplicate bug reports. This finding motivates the form of the textual similarity metric used by our algorithm.

Recall. Our second experiment was a direct comparison with the previous work of Runeson et al. [15]. In this experiment, each algorithm is presented with a known-duplicate bug report and a set of historical bug reports and is asked to generate a list of candidate originals for the duplicate. If the actual original is on the list, the algorithm succeeds. We perform no worse than the current state of the art.

Filtering. Our third and primary experiment involved on-line duplicate detection. We tested the feasibility and effectiveness of using our duplicate classifier as an on-line filter. We trained our algorithm on the first half of the defect reports and tested it on the second half. Testing proceeded chronologically through the held-out bug reports and predicted their duplicate status. We measured both the time to process an incoming defect report as well as the expected savings and cost of such a filter. We measured cost and benefit in terms of the number of real defects mistakenly filtered as well as the number of duplicates correctly filtered.

Features. Finally, we applied a leave-one-out analysis and a principal component analysis to the features used by our model. These analyses address the relative predictive power and potential overlap of the features we selected.

5.1 Textual Similarity

Our notion of textual similarity uses the cosine distance between weighted word vectors derived from documents. The formula used for weighting words thus induces the similarity measure. In this experiment we investigate the use of inverse document frequency as a word-weighting factor.

Inverse document frequency is a contextual measure of the importance of a word. It is based on the assumption that important words will appear frequently in some documents and infrequently across the whole of the corpus. Inverse document frequency scales the weight of each word in our vector representation by the following factor:

$$\mathsf{IDF}(w) = \log\left(\frac{\text{total documents}}{\text{documents in which word } w \text{ appears}}\right)$$

While inverse document frequency is popular in general natural language processing and information retrieval tasks, employing it for the task of bug report duplicate identification resulted in strictly worse performance than a baseline using only non-contextual term frequency.

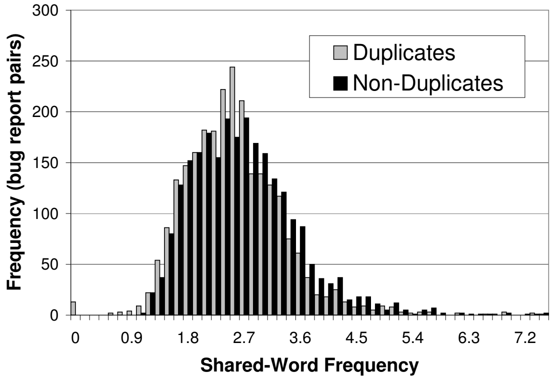

Figure 1. Distribution of shared-word frequencies between duplicate-original pairs (light) and close duplicate-unrelated pairs (dark).

We performed a statistical analysis over the reports in our dataset to determine why the inclusion of inverse document frequency was not helpful in identifying duplicate bug reports. First, we define the *shared-word frequency* between two documents with respect to a corpus. The shared-word frequency between two documents is the sum of the inverse document frequencies for all words that the two documents have in common, divided by the number of words shared:

$$\text{shared-word frequency}(d_1, d_2) = \frac{\Sigma_{w \in d_1 \cap d_2} \mathsf{IDF}(w)}{|d_1 \cap d_2|}$$

This shared-word frequency gives insight into the effect of inverse document frequency on word weighting.

We then considered every duplicate bug report and its associated original bug report in turn and calculated the shared-word frequency for the titles and descriptions of that pair. We also calculated the shared-word frequency between each duplicate bug report and the closest non-original report, with "closest" determined by TF/IDF. This measurement demonstrates that over our dataset, inverse document frequency is just as likely to relate duplicate-original reports pairs as it is to relate unlinked report pairs.

We performed a Wilcoxon rank-sum test to determine whether the distribution of shared-word frequency between duplicate-original pairs was significantly different than that of close duplicate-unique pairs. Figure 1 shows the two distributions. The distribution of duplicate-unique pair values falls to the right of the of distribution of duplicate-original pair values (with a statistically-significant p-value

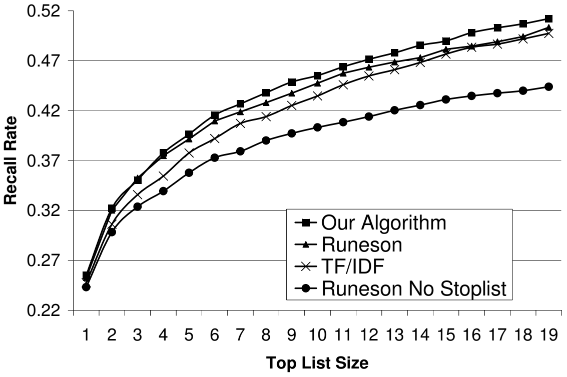

Figure 2. Recall rate of various algorithms on the Runeson task as a function of top list size. Our algorithm performs up to 1% better than that of Runeson et al., which in turn performs better than a direct application of inverse document frequency (TF/IDF).

of 4×10^{-7}). In essence, in the domain of bug report classification, shared-word frequency is more likely to increase the similarity of unrelated pairs than of than duplicate-original pairs. Thus, we were able to dismiss shared-word frequency from consideration as a useful factor to distinguish duplicates from non-duplicates. This study motivates the form of the weighting presented in Section 4.1.2 and used by our algorithm.

5.2 Recall Rate

Our second experiment applies our algorithm to a duplicate-detection recall task proposed by Runeson et al. [15] Their experiment consists of examining a corpus of bug reports that includes identified duplicate reports. For each duplicate report d, the algorithm under consideration generates an ordered *top list* containing the reports judged most likely to be d's original. This top list could then be used by the triager to aid the search for duplicate bug reports.

Results are quantified using a recall rate metric. If the true original bug report associated with d appears anywhere in the top list produced by the algorithm when given d, the algorithm is judged to have correctly recalled the instance d. The total recall rate is the fraction of instances for which this occurs. The longer the top list is allowed to be, the easier this task becomes. For every duplicate bug report, our algorithm considers every other bug report and calculates its distance to d using the word-vector representation and cosine similarity. The reports are then sorted and the closest reports become the top list. The algorithm of Runeson et al. is conceptually similar but uses a similarity metric with a different word weighting equation.

We tested four algorithms using all of the duplicate bug reports in our dataset as the corpus. Figure 2 shows the recall rates obtained as a function of the top list size. The TF/IDF algorithm uses the TF/IDF weighting that takes inverse document frequency into account; it is a popular default choice and serves as a baseline. As presaged by the experiment in Section 5.1, TF/IDF fared poorly.

We considered the Runeson et al. algorithm with and without stoplist word filtering for common words. Unsurprisingly, stoplists improved performance. It is interesting to note that while the TF/IDF algorithm is strictly worse at this task than either algorithm using stoplisting, it is better than the approach without stoplisting. Thus, an inverse document frequency approach might be useful in situations in which it is infeasible to create a stop word list.

In this experiment, our approach is up to 1% better than the previously-published state of the art on a larger dataset. We use our approach as the weighting equation used to calculate similarities in Section 5.3. However, the difference between the two approaches is not statistically significant at the $p \leq .05$ level; we conclude only that our technique is no worse than the state of the art at this recall task.

5.3 On-Line Filtering Simulation

The recall rate experiment corresponds to a system that still has a human in the loop to look at the list and make a judgment about potential duplicates; it does not speak to an algorithm's automatic performance or performance on non-duplicate reports. To measure our algorithm's utility as an on-line filtering system, we propose a more complex experiment. Our algorithm is first given a contiguous prefix of our dataset of bug reports as an initial training history. It is then presented with each subsequent bug report in order and asked to make a classification of that report's duplicate status. Reports flagged as duplicates are filtered out and are assumed not to be shown to developers directly. Such filtered reports might instead be linked to similar bug reports posited as their originals (e.g., using top lists as in Section 5.2) or stored in a rarely-visited "spam folder". Reports not flagged as duplicates are presented normally.

We first define a metric to quantify the performance of such a filter. We define two symbolic costs: *Triage* and *Miss*. *Triage* represents a fixed cost associated with triaging a bug report. For each bug report that our algorithm does not filter out, the total cost it incurs is incremented by *Triage*. *Miss* denotes the cost for ignoring an eventually-fixed bug report and all of its duplicates. In other words, we penalize the algorithm only if it erroneously filters out an entire group of duplicates and their original, and if that original is eventually marked as "fixed" by the developers (and not "invalid" or "works-for-me", etc.). If two bug reports both describe the same defect, the algorithm can save development triage effort by setting aside one of them. We assume the classifications provided by developers for the bug reports in our dataset are the ground truth for measuring performance.

We can then assign our algorithm a total cost of the form $a \times Triage + b \times Miss$. We evaluate our performance relative to the cost of triaging every bug report (i.e., $11,340 \times Triage$, since 11,340 reports are processed). The comparative cost of our model is thus $(11,340 - a) \times Triage + b \times Miss$.

In this experiment we simulated a deployment of our algorithm "in the wild" by training on the chronological first half of our dataset and then testing on the second half. We test and train on two different subsets of bug reports both to mitigate the threat of overfitting our linear regression model, and also because information about the future would not be available in a real deployment.

The algorithm pieces we have previously described are used to train the model that underlies our classifier. First, we extract features from bug reports. Second, we perform the textual preprocessing described in Section 4.1. Third, we perform separate pairwise similarity calculations of both title and description text, as in Section 4.1.1. This step is the most time intensive, but it is only performed during the initial training stage and when the model is regenerated.

Once we have calculated the similarity, we generate the graph structure for the clustering algorithm, as in Section 4.1.3. We use a heuristic to determine the best similarity cutoff for an edge to exist between two bug report nodes. For each possible cutoff, we build the graph and then examine how many edges exist between bug reports in an equivalence class and bug reports in different classes. We choose the cutoff that maximizes the ratio of the former to the latter. Given the cutoffs we can run the clustering algorithm and turn its output into a feature usable by our linear model.

We then run a linear regression to obtain coefficients for each feature. After the regression, we determine a cutoff to identify a bug as a duplicate. We do this by finding the cutoff that minimizes the *Triage*/*Miss* cost of applying that model as a filter on all of the training bug reports. In Section 5.3.1 we investigate how cutoff choices influence our algorithm's performance; in this section we present the results for the best cutoff.

Bug reports from the testing set are then presented to the algorithm one at a time. This is a quick process, requiring only feature recovery, the calculation of a total score based on the precalculated coefficients, and a comparison to a cutoff. The most expensive feature is the comparison to other reports for the purposes of the textual similarity features. The running time of this step grows linearly with the size of the historical set. In our experiments, the average total time to process an incoming bug report was under 20 seconds on a 3 GHz Intel Xeon. In 2005, the Mozilla project received just over 45,000 bug reports, for an average of one bug report every 12 minutes. Our timing results suggest our algorithm could reasonably be implemented online.

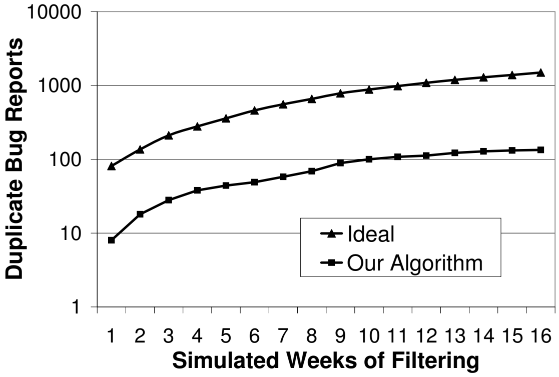

Figure 3. Cumulative number of duplicate bug reports filtered out as a function of the number of weeks of simulated filtering. Our algorithm never filtered out equivalence classes of reports that were resolved as "fixed" by developers. Note log scale.

Figure 3 shows the cumulative number of duplicate bug reports correctly filtered by our algorithm as a function of time. For comparison, the ideal maximum number of filterable bug reports is also shown. We correctly filtered 10% of all possible duplicates initially, trailing off to 8% as our historical information became more outdated. In this experiment we never regenerated the historical context information; this represents a worst-case scenario for our technique. At the end of the simulation our algorithm had correctly filtered 8% of possible duplicates without incorrectly denying access to any real defects.

Whether our algorithm's use as a filter yields a net savings of development resources generally depends on an organization's particular values for $Miss$ and $Triage$ — on other datasets our filter might incorrectly rule out access to a legitimate defect. Companies typically do not release maintenance cost figures, but conventional wisdom places $Miss$ at least one order of magnitude above $Triage$. In certain domains, such as safety-critical computing, $Miss$ might be much higher, making this approach a poor choice. There are other scenarios, for example systems that feature automatic remote software updates, in which $Miss$ might be lower. For *Triage*, Runeson et al. report on circumstance in which, "30 minutes were spent on average to analyze a [defect report] that is submitted" [15]. Using that figure, our filter

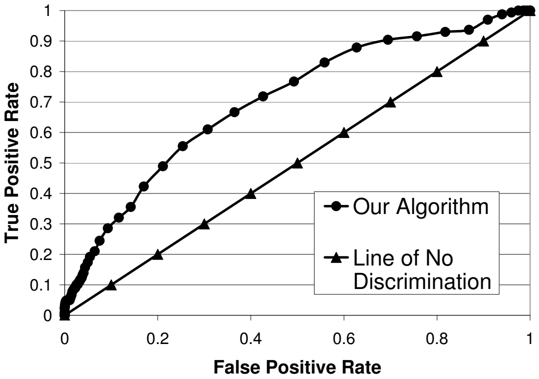

Figure 4. Receiver operating characteristic (ROC) curve for our algorithm on this experiment. A false positive occurs when all bug reports in an equivalence class that represents a real defect are filtered out. A true positive occurs when we filter one or more spurious reports while allowing at least one report through. Each point represents the results of our algorithm trained with a different cutoff.

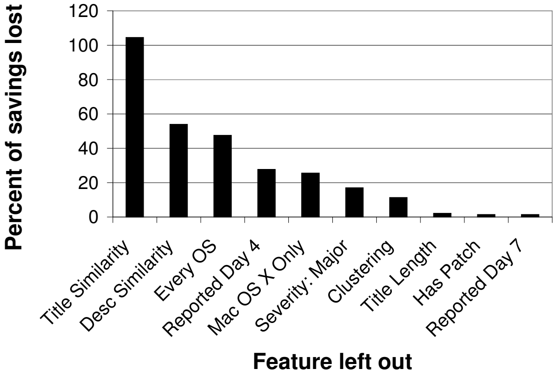

Figure 5. A leave-one-out analysis of the features in our model. For each feature we trained the model and ran a portion of the experiment without that feature. The y-axis shows the percentage of the savings (using *Triage* and *Miss*) lost by not considering that feature. Note that without the title textual similarity feature our model is not profitable in this simulation.

would have saved 1.5 developer-weeks of triage effort over sixteen weeks of filtering. We intentionally do not suggest any particular values for $Miss$ and $Triage$.

5.3.1 False Positives

In our analysis, we define a false positive as occurring when a fixed bug report is filtered along with all its duplicates (i.e., when an entire equivalence class of valid bug reports is hidden from developers). We define a true positive to be when we filter one or more bugs from a fixed equivalence class while allowing at least one report from that class to reach developers. A false positive corresponds to the concept of $Miss$ and an abundance would negatively impact the savings that result from using this system while a true positive corresponds to our system doing useful work.

In our experiment over 11,340 bug reports spanning four months, our algorithm did not completely eliminate any bug equivalence class which included a fixed bug. Hence, we had zero false positives and we feel our system is likely to reduce development and triage costs in practice. On this dataset, just over 90% of the reports we filtered were resolved as duplicates (i.e., and not "works-for-me" or "fixed"). While the actual filtering performed is perhaps modest, our results suggest this technique can be applied safely with little fear that important defects will be mistakenly filtered out.

We also analyze the trade off between the true positive rate and the false positive rate in Figure 4. When considering both safety and potential cost savings, we are most interested in the values that lie along the y-axis and thus correspond to having no false positives. However, this analysis suggests that our classifier offers a favorable tradeoff between the true positive rate and false positive rate at more aggressive thresholds.

5.4 Analysis of Features

To test our hypothesis that textual similarity is of prime importance to this task, we performed two experiments to measure the relative importance of individual features. We used a leave-one-out analysis to identify features particularly important in the performance of the filter. We also used a principal components analysis to identify overlap and correlation between features.

We perform the leave-one-out analysis using data saved from our modeling simulation. To measure the importance of each feature used in our filter, we reran the experiment without it. This included recalculating the linear model and retraining the relevant cutoff values, then re-considering each bug report, determining if the restricted model would filter it, and calculating our performance metric based on those decisions.

Once we obtain a performance metric for a restricted model, we can calculate the percent of savings lost as compared to the full model. The restricted models sometimes filter out legitimate bug reports; the $Miss$ cost must therefore be considered. We chose to illustrate this analysis with the values $Triage = 30$ and $Miss = 1000$. Figure 5 shows the change in performance due to the absence of each feature. Only features that resulted in more than a 1% performance decrease are shown.

The biggest change resulted from the feature corresponding to the similarity between a bug reports's title and the title of the report to which it was most similar. The second most important feature was the description similarity feature. This supports our hypothesis that semantically-rich textual information is more important than surface features for detecting duplicate defect reports. This had half the importance of the title, which is perhaps the result of the increased incidence of less-useful clutter in the description. Surface-level features, such as the relevant operating system or the conditions of the initial report, have less of an impact.

If these features are independent, a larger change in performance corresponds to a feature more important to the model. We performed a principal component analysis to measure feature overlap. This analysis is essentially a dimensionality reduction. For example, the features "height in inches" and "height in centimeters" are not independent, and a leave-one-out analysis would underrate the impact of the single underlying factor they both represent (i.e., since leaving out the inches still leaves the centimeters available to the model). For our features on the training dataset there were 10 principal components that each contributed at least 5% to the variance with a contribution of 10% from the first principal component; it is not the case that our features strongly intercorrelate.

6. Conclusion

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. We empirically evaluated our approach using a dataset of 29,000 bug reports from the Mozilla project, a larger dataset than has generally previously been reported. We show that inverse document frequency is not useful in this task, and we simulate using our model as a filter in a real-time bug reporting environment. Our system is able to reduce development cost by filtering out 8% of duplicate bug reports. It still allows at least one report for each real defect to reach developers, and spends only 20 seconds per incoming bug report to make a classification. Thus, a system based upon our approach could realistically be implemented in a production environment with little additional effort and a possible non-trivial payoff.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

[3] B. Boehm and V. Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.

[4] G. Canfora and L. Cerulo. How software repositories can help in reoslving a new change request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.

[5] D. Čubranić and G. C. Murphy. Automatic bug triage using text categorization. In *Software Engineering & Knowledge Engineering (SEKE)*, pages 92–97, 2004.

[6] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

[7] M. Kantrowitz, B. Mohit, and V. Mittal. Stemming and its effects on TFIDF ranking. In *Conference on Research and development in information retrieval*, pages 357–359, 2000.

[8] S. Kim and J. E. James Whitehead. How long did it take to fix bugs? In *International workshop on Mining Software Repositories*, pages 173–174, 2006.

[9] H. Liu. MontyLingua: an end-to-end natural language processor with common sense. Technical report, `http://web.media.mit.edu/~hugo/montylingua`, 2004.

[10] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. Clustering social networks. In *Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, pages 56–67, 2007.

[11] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. In *Conference on Requirements Engineering*, pages 283–294, 2004.

[12] C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[13] E. S. Raymond. The cathedral and the bazaar: musings on linux and open source by an accidental revolutionary. *Inf. Res.*, 6(4), 2001.

[14] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the Mozilla project. In *Open Source Software Development Workshop*, pages 155–175, 2002.

[15] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *International Conference on Software Engineering (ICSE)*, pages 499–510, 2007.

[16] J. Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[17] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Workshop on Mining Software Repositories*, May 2007.