

Modeling Bug Report Quality

Pieter Hooimeijer and Westley Weimer
University of Virginia
Charlottesville, VA. 22903
{pieter, weimer}@cs.virginia.edu^{*}

ABSTRACT

Software developers spend a significant portion of their resources handling user-submitted bug reports. For software that is widely deployed, the number of bug reports typically outstrips the resources available to triage them. As a result, some reports may be dealt with too slowly or not at all.

We present a descriptive model of bug report quality based on a statistical analysis of surface features of over 27,000 publicly available bug reports for the Mozilla Firefox project. The model predicts whether a bug report is triaged within a given amount of time. Our analysis of this model has implications for bug reporting systems and suggests features that should be emphasized when composing bug reports.

We evaluate our model empirically based on its hypothetical performance as an automatic filter of incoming bug reports. Our results show that our model performs significantly better than chance in terms of precision and recall. In addition, we show that our model can reduce the overall cost of software maintenance in a setting where the average cost of addressing a bug report is more than 2% of the cost of ignoring an important bug report.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Debugging aids*; D.2.8 [Software Engineering]: Metrics; D.2.9 [Software Engineering]: Management—*Life cycle*; D.2.9 [Software Engineering]: Management—*Time estimation*

General Terms

Economics, Experimentation, Human Factors, Management, Measurement

^{*}This research was supported in part by National Science Foundation Grants CNS 0627523 and CNS 0716478 and Air Force Office of Scientific Research grant BAA 06-028, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.
Copyright 2007 ACM 978-1-59593-882-4/07/0011 ...\$5.00.

Keywords

bug report triage, issue tracking, statistical model, information retrieval

1. INTRODUCTION

A significant portion of overall software development is spent addressing defects. Boehm and Basili claim that maintenance consumes over 70% of the total lifecycle cost of a software product [5]. Modifying existing code, dealing with defects, and otherwise evolving software are major parts of that maintenance [12]. Large projects often use bug reporting and triage systems to cope with defect reports [2]. However, the number of reports typically exceeds the resources available to address them; rather than having the development resources to deal with every defect, even mature software projects are forced to ship with both known and unknown bugs [10]. A lack of resources often constrains developers to deal with some bug reports too slowly or not at all.

A further complication is that many submitted bug reports may be spurious duplicates or descriptions of non-defects. Previous studies have found that as many as 36% of bug reports were duplicates or otherwise invalid [3]. The triage work in evaluating bug reports consumes developer time and effort [14]. Bug report triage and evaluation are a significant part of modern software engineering for many large projects.

In this paper we attempt to reduce bug report triage costs by separating the wheat from the chaff. We present a model of bug report quality that predicts whether developers will choose to address a bug report, by measuring whether the report is addressed within a given amount of time. We do this by classifying bug reports as either “cheap” or “expensive” to triage. For the purposes of this paper, *triage* is the act of inspecting a bug report, understanding its contents, and making the initial decision regarding how to address the report.

Our model is based on features that can easily be gathered from bug report submissions, without referring to past bug reports. We base our model on a statistical analysis of over 27,000 bug reports from the Mozilla Firefox project, and we experimentally validate its predictive power. Our model can be used by developers to filter incoming bug reports or to aid with bug report prioritization.

In practice, Mozilla and many other open source software projects make use of bug reporting and triage software [2] that is open to the public. The assumption is that allowing users to report and potentially help fix bugs improves

overall quality [13]. These reporting systems allow users to report, track, describe, comment on and classify bug reports and feature requests. Bugzilla is a particularly popular open source bug reporting software system 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, which often take the form of patches or screenshots.

Bugzilla enforces a specific work flow for each bug report. Reports start out as *unconfirmed* and remain that way until quality assurance is able to reproduce the bug. Bug reports can pass through several other stages repeatedly before finally being *resolved*. Bug reports that are *resolved* receive one of the following designators: *duplicate*, *invalid*, *fixed*, *wontfix*, or *worksforme*. These indicate why the report was closed; for example, *worksforme* indicates that quality assurance was unable to reproduce the issue described in the report.

Our model relates bug report attributes to the final status of that bug report. It can be used as a filter that sits between incoming bug reports and developers and sets aside bug reports that are not of interest, for example because they will never be resolved. Such a filter is useful if its benefits outweigh its costs. In this case the benefits consist of saved triage effort, and the costs are associated with legitimate defects for which all sources of information are filtered away. For our data set, our model is sufficiently precise that it can reduce overall software maintenance costs if the average cost of addressing a bug report is more than 2% of the cost of ignoring an important bug report—a ratio that we claim is quite reasonable in practice.

In addition, the relative weights of various features in our model suggest changes to bug reporting software and triage formats. For example, our model suggests that the *severity* field has a significant effect on the lifetime of the bug. We also found that the model places significant weight on the number of comments posted within the first hour of a report's lifetime. This may indicate that reports frequently need to be clarified, or that bug reports are more likely to be addressed if more users express an interest.

The main contributions of this paper are:

- A model of bug report quality based on features available when the bug report is submitted. Over our data set, the model has reasonable predictive power as an automatic filter, and can reduce the overall cost of software maintenance in a setting where the average cost of addressing a bug report is more than 2% of the cost of ignoring an important bug report.
- A discussion of the features involved in the model and their implications for bug reporting and triage systems.

The structure of this paper is as follows. In Section 2 we describe common bug reports and present a motivating example linking bug report features and final bug report resolutions. In Section 3 we formalize a model for bug report quality. Section 4 presents the experiments that measure our model's ability to predict whether a bug is triaged within a given amount of time. In Section 5 we analyze the results.

Section 6 compares our work to similar research projects and Section 7 concludes and gives future directions.

2. MOTIVATING EXAMPLE

During the period 2003-2006, the Mozilla project received over 140 bug reports about the download status indicator, which incorrectly showed a negative file size when downloading files larger than 2 GB. This bug occurred in several Mozilla products, but most reports cited Firefox and its predecessor, Firebird, as exhibiting the problem.

Mozilla bug report 221359, which we will call *First*, was filed in October 2003 describing this defect. The bug report included a screenshot demonstrating the issue, and reported a severity of *minor*, but gave potentially-misleading steps to reproduce the problem: "Put a 2GB+ file on an IIS 5.0 based Web Server [...] Download the 2GB+ file on a Windows XP based system using Firebird." Within fifty minutes a developer had commented on the bug report by asking for addition details: "does this happen on other webservers? does this occur on Mozilla as well?" Lacking addition comments from others, the developer changed the report status to *worksforme* three weeks after the initial filing.

Mozilla bug report 228968, which we will call *Second*, was filed ten weeks later describing the same defect. The report listed the severity as *normal*, did not include a screenshot, noted a Windows NT base system, and included the text: "Downloading a large file (about 3GB) displays negative values for file size (Status:xxxKb of -1173483Kb at xxxKb/sec), after a while the speed becomes negative (-xxxKB/sec)." The bug was initially marked as a *duplicate* of bug 184452, which covers upgrading the project's core generic input stream interfaces to handle 64-bit file access, and was eventually reverted back to an *unconfirmed* status with a dependency on bug 184452. All of this activity took place within four days of the report's initial submission. Over the course of several months, a number of users and developers commented on this report, confirming it and supplying screenshots.

The *Second* bug report was fixed a year later, in April 2005. One month later, *First* was resolved as a *duplicate* of *Second*. This example helps to show the difficulties involved in bug report triage in large projects, even when bug tracking software systems are used. Bug report status changes exhibit a more complex behavior than one might expect.

Beyond the importance of the underlying defect, which is the same for these two duplicate bug reports, the reports have a number of features that may influence their eventual status and the time taken to arrive at it. These two reports differ in their self-reported severity (*minor* vs. *normal*), host operating system (Windows XP vs. Windows NT), number of comments (*few* vs. *many*), attachments (*screenshot* vs. *nothing*), and possibly their descriptive quality (*potentially misleading* vs. *direct*). One might imagine that the initial screenshot provided by the *First* report would prevent it from being dismissed as *worksforme*, but that was not the case. In practice, the *Second* bug report gained the attention and was eventually fixed directly, possibly because of its comment count and self-reported severity.

This single example is not sufficient to tease apart the interplay between bug report features such as severity, clarity, attachments, and comment count and their influence on the report's final resolution time. The *First* bug report had a ten week lead time on the *Second*, but developers were un-

able to take full advantage of its early warning because of the **workforme** status assignment. Since intuition about this process is difficult and the triage and resolution process is complicated, we desire a formal model relating bug report features and bug report resolution time.

3. A MODEL OF BUG REPORT QUALITY

We want to produce an model of bug report quality that reduces the cost of bug report triage and is based on objective and easy-to-gather features. Unfortunately, per-report cost data is not usually available; while it remains a topic of recent research, a general formula for developer time costs has yet to be established. Instead, we measure how long a report takes to move from its initial **unconfirmed** state to one of the **resolved** states. This approximation to the true triage cost is based on the assumption that, in general, reports of higher quality are dealt with more quickly than those of lower quality. In other words, we assume that the “time until resolved” is a good indicator of how expensive a bug report was to triage.

Different classes of bug reports undergo different steps while moving from **unconfirmed** to **resolved**. In the Firefox project used in our experiments, for example, a bug report marked as **fixed** must be associated with a change to the source code. Developing a patch would take time, regardless of how good the initial bug report is. In contrast, a bug report that is marked as a **duplicate** may be resolved in minutes if the triager is already familiar with the problem.

This difference constitutes a possible confound,¹ since some types of reports may consistently be resolved faster than others. It is, therefore, tempting to include “resolution type” as an input to the model. However, our goal is to assist as early as possible during bug report triage, which precludes using features that are not known until a report is resolved. Instead, we model “time until resolved” as a boolean value. If a report is resolved within a certain cutoff, it is categorized as “cheap to triage.” If not, it is marked as “expensive.”

This construction is based on the additional assumption that, for some cutoff, bug reports that take longer to resolve must have been expensive to triage, regardless of their resolution type.

We use a basic linear regression model to classify bug reports as either cheap or expensive to triage. Recent work in automatic bug report analysis has tended to use sophisticated machine learning techniques, such as support vector machines [3] and clustering [17], in combination with text categorization techniques [4]. An important advantage of linear regression over other techniques is that the resulting models are straightforward to analyze. We use analyses of variance to assess the relative contribution of bug report features to an accurate prediction.

3.1 Model Features

Our model classifies bug reports based on surface features that can be extracted from a report within the first few days after it is submitted. This excludes any features that require comparing the report to previous reports, such as textual similarity. We use this restriction because textual categorization, such as the method used by Weiß et al. [17]

¹A “missing variable” in the model, the effect of which is incorrectly attributed to error rather than the effect of the variable.

is computationally much more expensive than a basic linear model, especially as the number of bug reports grows.

3.1.1 Self-Reported Severity

When filing a report using the Bugzilla bug tracking system, the user is asked to rate the bug’s severity using one of the following: **blocker**, **critical**, **major**, **normal**, **minor**, **trivial**, or **enhancement**.

We treat this variable as an ordered factor, where **blocker** is the most severe and **enhancement** is the least severe. One problem with self-reported severity is that users might not follow the guidelines describing each category. It may be tempting, for example, for the user to over-report a bug’s severity to have it looked at more quickly. To test this hypothesis, we also monitor subsequent changes to the severity field. Section 3.1.5 describes this in more detail.

3.1.2 Readability Measures

Our model incorporates several basic readability measures. These include the Coleman-Liau formula, the Kincaid formula, the Automated Readability Index, and the SMOG index [11], all of which are based on surface features such as the average number of syllables per word or words per sentence. We hypothesize that bugs that are more difficult to understand will be more difficult to deal with and will be addressed later. These measures are applied to the initial bug report description using GNU Style version 1.10-rc4.

3.1.3 Daily Load

It is possible that bug reports may be dealt with more slowly if there is a large influx of other bug reports at around the same time. This might occur following a new release, for example. We account for this by associating with each bug report the total number of bugs submitted in the 24 hours preceding and following the submission of that report.

3.1.4 Submitter Reputation

For each bug report, we define its submitter’s “reputation” as follows:

$$\text{reputation} = \frac{|S \cap R|}{|S| + 1}$$

Where S is the set of all reports *submitted* by that submitter before the bug report that is under consideration, and R is the set of all bug reports that were either **resolved** or marked as the duplicate of a **resolved** report. In other words, this score measures the submitter’s “success rate” prior to submitting the report under consideration.

3.1.5 Changes over time

Our model includes several features that are measured after the initial report is submitted. These features include:

- **bug severity changes**—We hypothesize that bug reports with overstated severity take longer to fix. In order to test this hypothesis, the model includes the total number of severity escalations and de-escalations within a set of given time periods.
- **comment count**—At each given time offset, we record the total number of comments that have been posted in response to the initial bug report. Note that, in general, comments can be posted by anyone. This means that comment count could serve as a proxy for some notions of popularity.

- attachment count—At each given time offset, we record the total number of attachments that are associated with the current report. In previous work, we presented results that suggest that bug reports that have a patch included are dealt with more quickly [16]. We distinguish between patches and other types of attachments, such as screenshots.

In our experiments, we examine how much post-submission data is needed to yield adequate predictive power. We recorded this data for several fixed time offsets following the report’s submission, and used it in various different ways, as described in Section 4.3.

3.1.6 Elided Features

In addition to the aforementioned features, we considered bug priority changes in the same way as severity changes. Priority changes are typically internal bookkeeping annotations made by a developer once the bug has been assigned for further analysis. Priority changes turned out to be very infrequent, however, and the effect of these features was negligible. Similarly, we do not include fields such as the operating system and product version used. We assume that the bug reporter simply fills these out truthfully, making these features mostly irrelevant to the overall quality of the report.

4. EXPERIMENTS

In the following sections, we present several experiments designed to:

- **validate the assumptions** that underlie our model, such as whether it is reasonable to use a bug report’s lifetime as a proxy for how expensive it was to triage. In Section 4.2, we do so by looking at the distribution of bug report lifetimes between **unconfirmed** and **resolved** for different resolution types.
- **find how much post-submission data is necessary** to make accurate predictions. In Section 4.3 we test our hypothesis that useful predictions can be made from features of a bug report, such as the number of comments and the number of attachments, that are available early after its submission.
- **find the optimal “resolved by” cutoff** with respect to the the F_1 -score, which we describe in Section 4.1. In Section 4.3, we examine the optimal cutoff by evaluating the model’s performance for cutoffs ranging from 25 to 55 days.
- **evaluate the hypothetical benefit** of our model if used as a filter. The premise in this experiment is that the model is given exclusive control over which bug reports are reviewed by a human developer. The experiment, described in Section 4.4, is designed to find the cost ratio between false positives and false negatives at which the model becomes cost-effective.

The next section describes the general data acquisition and conversion for the experimental data that follows.

4.1 Experimental Design

Our experiments are based on a data set of 51,154 bug reports for the Mozilla Firefox project. This data was supplied

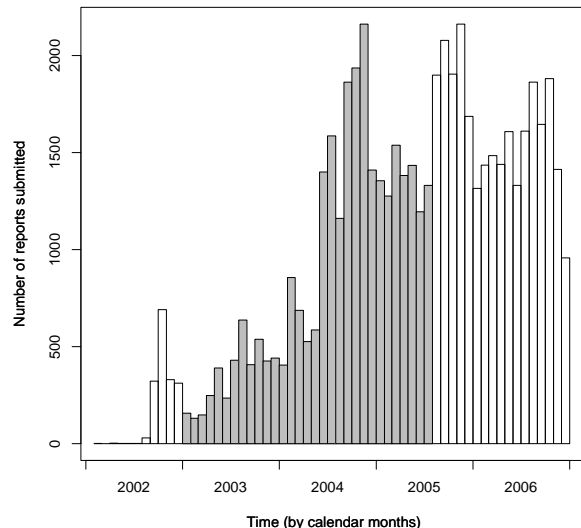


Figure 1: The number of bug reports filed per month for the Firefox project. The shaded bars indicate the data used in our experiments.

by the Mozilla project and verified informally against data obtained from Mozilla’s Bugzilla website using a crawling tool. We chose the Firefox project primarily because of its size, wide deployment, wide range of bug report submitters, and the public availability of its bug reports.

Figure 1 shows the frequency of bug report submissions in the period 2002–2006. The reports in our data set were filed between 07/30/1999 and 12/26/2006. We used slightly more than half of the reports, all filed between 01/01/2003 and 07/31/2005. The first cutoff was selected based on the fact that, prior to 2003, the frequency of bug report submissions was significantly lower than in the period 2003–2006. The second cutoff was chosen to coincide with the project’s release cycle, as Firefox 1.5 beta testing started in September 2005. From the resulting data set we eliminated another 293 bug reports based on missing readability scores (e.g., caused by empty descriptions). Our final data set consists of 27,984 bug reports.

4.2 Experiment 1 – Validating the model

This experiment documents our initial exploration of the Firefox bug report data set, which prompted the design decisions outlined in Section 3. Figure 2 shows the relative frequency of bug report lifetimes by their eventual resolution type. As noted in Section 3, we want our model to employ a single time-to-resolve cutoff value across reports in all these categories. Reports that are not resolved by this cutoff are classified as “expensive,” while those that do meet the deadline are classified as “cheap.”

Figure 3 shows percentiles for report lifetime. Reports with resolution type **workforme** and **fixed** cover a significantly longer range of lifetimes than, for example, duplicates. The fastest 80% of duplicates are resolved within 45 days, while the fastest 80% of reports marked **workforme** are resolved within 235 days. In addition to the data presented

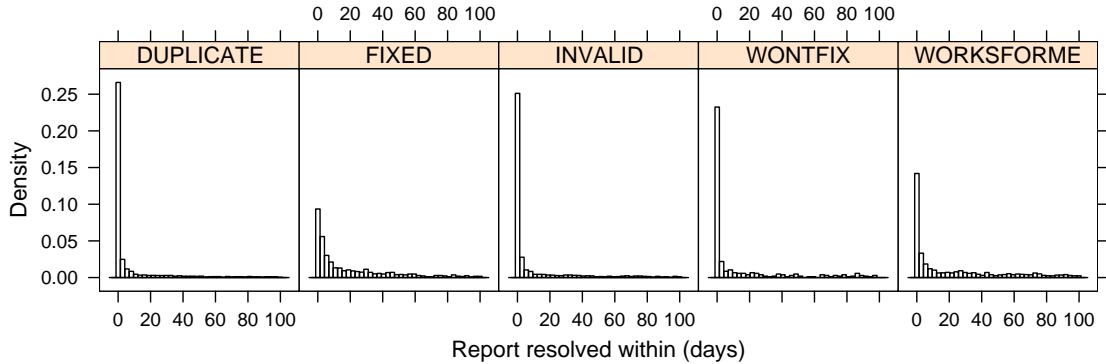


Figure 2: The relative frequency of bug reports resolved in 2.5 day increments, by final status. While the distributions are not identical, they are all strongly left-heavy, which suggests that it is reasonable to use a single cutoff value.

Type	Percentile				Freq.
	50th	60th	70th	80th	
duplicate	0.4	1.2	7.3	45.0	40%
fixed	29.0	53.7	113.2	211.1	12%
invalid	0.7	2.1	14.9	62.6	18%
wontfix	10.9	49.3	123.7	243.9	4%
worksforme	58.0	102.5	157.0	235.3	26%

Figure 3: The number of days it took for bug reports to be resolved, for the given percentiles, by resolution type. For example, 12% of all resolved reports were eventually fixed, and of that 12%, half were resolved within 29 days.

in this figure, our data set contained 2,675 bug reports that were never resolved, accounting for 9.6% of our total data. These bugs are always marked as not having met the cutoff.

This data leads to two meaningful conclusions. First, there are significant differences in the distribution of bug report lifetimes. Second, all of the distributions are left-skewed. This is promising since it may allow a single cutoff to determine with reasonable accuracy which reports took unusually long to resolve. In Section 4.3, we evaluate the performance of our model for several different cutoff values.

4.3 Experiment 2 – Model Selection

In this experiment, we evaluate the performance of our model based on two parameters:

- The amount of post-submission data, such as comment count and attachment count, that the model is allowed to use. The goal is to find the minimum amount necessary for the model to yield adequate performance.
- The cutoff used to classify bugs as “cheap” or “expensive”. In Section 4.2, we established that there are significant differences in the distribution of bug report lifetimes for reports that have different resolution types. In this experiment we try to find the optimal cutoff, and we verify whether our performance is significantly better than chance.

4.3.1 Precision and Recall

We evaluate our performance results in terms of *precision* and *recall*, two measures commonly used in information retrieval [15]. In this context, we treat our model as a document retrieval system that returns bug reports based on the query: “Which reports were resolved before the cutoff?” The performance measures are defined as follows:

$$\text{precision} = \frac{|C \cap R|}{|R|} \quad \text{recall} = \frac{|C \cap R|}{|C|}$$

where C is the *correct* set of bug reports, and R is the set of reports returned by the model.

Precision and recall measure different types of performance. Intuitively, recall measures what portion of relevant bug reports our model was able to find. Conversely, precision measures what portion of the returned results actually satisfied the query. In other words, recall penalizes false negatives, and precision penalizes false positives.

Either returning all bug reports and returning a single correct bug report would trivially maximize recall and precision respectively. To avoid favoring such degenerate models, precision and recall can be combined into a single-valued F -score, which is their weighted harmonic mean:

$$F_\alpha = \frac{(1 + \alpha)pr}{\alpha p + r}$$

where p is precision and r is recall. The α terms represent the relative weighting of precision over recall. For the following experiments we use $\alpha = 1$, giving precision and recall equal weight.

4.3.2 Cross Validation

Linear least squares regression is an instance of a supervised learning algorithm. When applied directly, the model is being trained and tested on bug reports from the same data set. Although the data set is large enough to yield statistically significant results, the possibility exists that our subsampling of bug reports is not completely representative of Firefox bugs in general. Consequently, a direct training method might cause the model to produce overfit, yielding results that do not generalize well outside of the data set that was used.

We use 10-fold cross validation [9] to detect overfitting.

In this procedure, the elements in the data set are randomly assigned to ten groups of approximately equal size. The model is trained and evaluated ten times. Each group is held out once for testing, using the remaining nine groups to train the model. This way, the model is never trained and tested on the same data. After this, the cross validation results can be averaged and compared to the same model when trained and tested using the entire data set. Any differences in outcome are referred to as *bias*, and significant bias would be suggestive of overfitting.

4.3.3 Experimental Procedure

For each bug report in the data set, we compute the post-submission features using Bugzilla’s activity log. We record the value for each feature at eight deadlines following submission: 1 hour, 12 hours, 1 day, and 2–10 days in 2-day increments. We construct two different models for each post-submission deadline:

- A *cumulative* model. For each submission deadline, the corresponding model uses the post-submission data from that deadline and each preceding deadline. For example, the model for the 12-hour deadline uses both the 12-hour data and the 1-hour data.
- A *non-cumulative* model. The model corresponding to each deadline uses only the post-submission data recorded for that deadline.

We classify the bug reports based on seven resolved-by cutoff times: 25–55 days in 5-day increments. The feature corresponding to each cutoff is set to 1 if the bug was resolved before the cutoff, and to 0 otherwise. For each combination of submission deadlines and resolution cutoffs, we train the two alternative models as follows:

1. We first perform the cross-validation steps. The model is trained as if the response variable were continuous in the range $[0, 1]$.
2. The model outputs are turned into a classification by rounding. We perform a linear search to find a *model cutoff* in the range $[0, 1]$. Reports with a score greater than the cutoff receive a 1; all others receive a 0. We record the model cutoff that yields the highest F_1 -score for this validation step.
3. After performing the cross validation steps, we train the model on the entire data set. We compute the average model cutoff from the cross validation steps, and use that to compute precision, recall, and F_1 -score for the model over the entire data set.

Note that the average model cutoff from the cross validation steps need not be the optimal cutoff for the model trained on the whole data set. We avoid calculating this optimal cutoff because it is more prone to cause the classification to overfit to our data set.

4.3.4 Results

To evaluate this set of models, we first compute the performance of a degenerate model to serve as a lower bound. For each resolution cutoff in the range 25–55 days, the number of reports that were resolved before the deadline is greater than the number of reports that were not. This means it is reasonable to consider a model that outputs 1 for every

cutoff (days)	Performance	
	Precision	F_1 -score
25	0.56	0.72
30	0.57	0.73
35	0.58	0.74
40	0.59	0.74
45	0.60	0.75
50	0.61	0.76
55	0.62	0.76

Figure 4: Performance for a degenerate model that always returns “this bug report will be resolved before the cutoff.” This model is equivalent to the current practice of triaging every submitted bug report.

input. In addition, if the model is used as a filter, as in Section 4.4, this corresponds to looking at every bug report, which is the current industrial practice.

Such a model achieves perfect recall, since it always returns all relevant results. Precision would be exactly equal to the ratio of reports that met the deadline to the total number of reports in the data set. Figure 4 shows the precision and F_1 -score for each resolution cutoff.

Figure 5 shows precision, recall, and F_1 -score for the cumulative model (left panel) and the non-cumulative model (right panel), for a resolution cutoff of 30 days. The grey lines indicate the F_1 -score and precision for the degenerate model. The F_1 -score for the two models is slightly better than that for the degenerate model. The maximal improvement over the degenerate model is 0.05, achieved by the cumulative model using 10 days worth of post-submission data and a resolution cutoff of 25 days.

It is interesting to note that the non-cumulative model shows diminishing performance for decision deadlines greater than 1 day (see Figure 5). The cumulative model, on the other hand, shows more or less level performance for the same range of deadlines. We hypothesize that the cumulative model performs better because it uses the “early” post-submission features, which the non-cumulative model does not have access to.

Figure 6 illustrates the full set of F_1 -score results for the cumulative model. Moving along the vertical axis, performance improves significantly between 60 minutes and one day of post-submission data. Performance is mostly level when using more than one day of data. The model performs slightly better for larger resolution cutoffs. This is also true for the degenerate model, however.

Figure 7 shows the results of a per-feature analysis of variance on the cumulative model, using six days worth of post-submission data and a resolution cutoff of 30 days. The table lists only those features with a significant main effect ($\alpha = 0.05$). F denotes the F -ratio, which is close to 1 if the feature does not affect the model, while p denotes the significance level of F (i.e., the probability that the feature does not affect the model).

Self-reported severity does not have a single-valued coefficient. The severity feature only has eight discrete possible values ranging from *enhancement* to *blocker*, to which we assigned numerical values 1–8. The relationship between these severity levels might not be linear, however, so we included

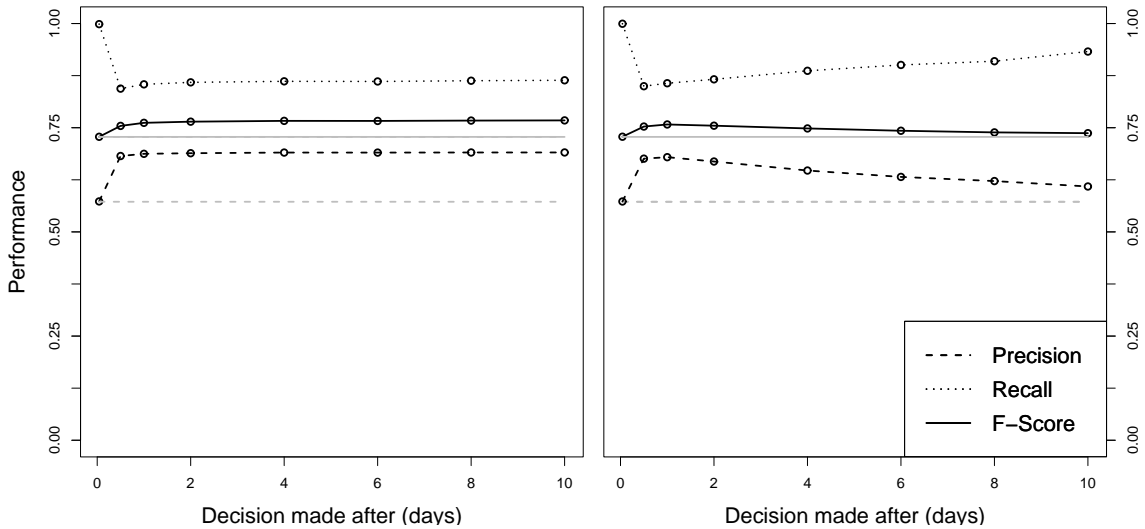


Figure 5: Performance using time-dependent data cumulatively (left) and using a single snapshot at the decision deadline (right). The light lines denote the F_1 -score and precision for the degenerate model.

six polynomial transformations of this feature (i.e., powers 1–6). As a result, there is no single coefficient for severity. The polynomial is monotonically increasing over the range [1; 8], however, evaluating to 0.017 for enhancement and 0.14 for blocker. This shows that a higher severity rating correlates with a shorter bug report turnaround.

The analysis of variance shows that comment count and attachment count are the most important post-submission features, especially within the first day. Self-reported severity also has a significant effect, but severity changes (recorded as a post-submission feature) do not. Variance analyses performed for the other cumulative models show similar results.

Our cross-validation steps revealed little to no bias. The maximum absolute difference in F_1 -score between a model and its corresponding cross validation steps was 0.00259 for the cumulative model and 0.00343 for the non-cumulative model. This shows that the averaged results from each set of cross validation steps were never significantly different from the results obtained using the corresponding model trained on the entire data set.

4.4 Experiment 3 – Performance Analysis

In Section 4.3, we evaluated our model’s performance based on its predictive power relative to a degenerate model. Our basic claim, however, is that our model can be used to reduce the cost of software maintenance by filtering out bug reports that are “expensive” to triage. In this experiment, we explicitly evaluate our model in terms of its cost compared to the degenerate model, which represents the current practice of triaging all bug reports.

4.4.1 Experimental Procedure

We define two symbolic costs: *Triage* and *Miss*. *Triage* represents a fixed cost associated with triaging a bug report. For any bug report that the model does not filter out, the total cost incurred by the model is incremented by *Triage*. *Miss* denotes the cost for ignoring a fixed bug report and all of its duplicates. In other words, if one or more bug report in a group of duplicates is marked as fixed, then we penalize

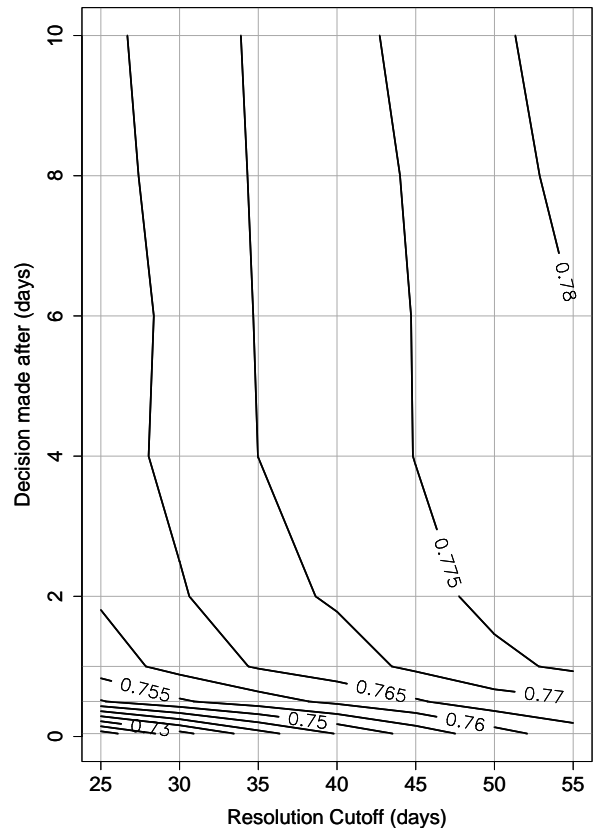


Figure 6: F_1 -score for the cumulative model, for each combination of resolution cutoff (horizontal axis) and the amount of post-submission data used (vertical axis). The grid lines indicate the sampling points. Performance levels off when using more than one day of post-submission data.

Feature	Coefficient	F	p
Severity	N/A	11.7	0
Coleman-Liau	-0.0063	11.0	0
Daily Load	-0.00069	135	0
# Comments (60 min.)	0.063	196	0
# Comments (12 hrs.)	0.050	94.2	0
# Attachments (1 day)	-0.12	10.4	< .001
# Attachments (6 days)	-0.13	7.34	< .001
ARI	-0.0086	4.87	< .05
# Attachments (60 min.)	-0.051	6.40	< .05
# Comments (4 days)	0.018	6.58	< .05

Figure 7: Analysis for variance for the cumulative model, using 6 days of post-submission data and a 30-day resolution deadline. Features with no significant main effect ($p \geq 0.05$) are not shown.

Cutoff (days)	# Triage	# Misses	Ratio
30	19,832	186	0.023
35	20,512	167	0.022
40	20,721	158	0.022
45	21,988	125	0.021
50	21,801	135	0.021
55	23,443	92	0.020

Figure 8: Evaluation data for the cumulative model using one day of post-submission data. Only bugs that the model predicts will be resolved before the cutoff are triaged. If $Triage/Miss > Ratio$ for a particular setting then using our model as a filter may save development resources in that setting.

the model only if it erroneously filters out the entire group. If two bug reports both describe the same defect, the model can save development triage effort equal to $Triage$ by setting aside one of them.

For this experiment, we use the models generated in Experiment 2. The models are trained to optimize for F_1 -score, as before. For each model, we compute total cost, which is of the form $aTriage + bMiss$. We evaluate our performance relative to the cost of triaging every bug report. That cost is $27,984 \times Triage$, since our data set contains 27,984 reports. This means that the comparative cost of a model M is $(a_M - 27,984) Triage + b_M Miss$. For each model, we compute the ratio $Triage/Miss$ that would be required for that model to outperform the degenerate model. We assume that $Miss \gg Triage$, that is, that the cost of missing an important bug far exceeds the cost of triaging a bug report, so we would like this ratio to be as small as possible.

4.4.2 Results

The best-performing model achieved a ratio of 0.0195, using eight days worth of post-submission data and resolution cutoff of 55 days. This means that our model out-performs the degenerate model if the cost of $Triage$ is greater than 1.95% of the cost of a $Miss$.

In practice, allowing the model to wait for eight days would be impractical. Fortunately, the model has adequate performance using just one day to decide. Figure 8 shows the number of triages, misses, and the cost ratio for several

models using a single day of post-submission data.

5. DISCUSSION

Our experiments show that a relatively unsophisticated linear model yields better-than-chance predictive power. In Experiment 2, we show that such a model achieves significantly better precision than the degenerate case. The model’s performance suggests that it does a reasonable job of modeling whether a bug report is eventually addressed. This means that the model captures some notion of the quality of bug reports, and also the importance of the defect that is described. We interpret our model’s features accordingly.

Our analysis of these models shows an interesting trend: “early” features, in particular attachment count and comment count, matter most. Interestingly, attachment count has a negative coefficient whenever its effect is significant. In other words, the presence of an attachment with a bug report is correlated with a higher probability of being marked as “expensive.” Comment count, on the other hand, has a positive coefficient whenever its effect is significant. This might suggest that bugs that receive more user attention get fixed faster, or that important bugs that are fixed quickly receive a great deal of user attention.

The Coleman-Liau and Automated Readability indices both attempt to approximate how many years of education is necessary to comprehend the text, measured in US grade level. Both indices receive negative coefficients, indicating that easier-to-read descriptions correlate with a short bug report lifetime.

Several features proved not to have a significant effect on the model’s output. Among these were patch count, which we hypothesized would have a significant effect in previous work [16]. Similarly, the numbers of severity escalations and de-escalations were not found to be consistently significant for the various models. A possible explanation for this is that these features are relatively infrequent; for example, only around 900 bugs in our data set had a nonzero patch count within a day.

In Experiment 3, we tested our model’s usefulness as a filter. Whether such a filter represents a net savings of development resources depends on that organization’s particular values for $Miss$ and $Triage$. In addition, the value for $Miss$ may vary significantly depending on the particular defect. Companies typically do not release maintenance cost figures, but conventional wisdom places $Miss$ at least one order of magnitude above $Triage$.

Specific values that we obtained from software development divisions at IBM and Microsoft were made available on the condition that we not publish them. For the purposes of comparison, however, if $Triage$ is \$30 and $Miss$ is \$1000, using our model as a filter saves between five and six percent of the development costs for this data set. 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. We intentionally do not suggest any particular values for $Miss$ and $Triage$, instead presenting the required ratio at which our proposed technique becomes profitable.

6. RELATED WORK

Anvik et al. use support vector machines and text cate-

gorization to automatically assign bug reports to an appropriate developer. They claim that their system could aid a human triager by recommending a set of developers for each incoming bug report [3]. It would be interesting to gauge if their algorithm’s performance is correlated with the difficulty of triaging a bug report.

Kim and Whitehead measure the time to fix a bug in two software projects and claim that bug fix time is a useful measure of software quality [8]. By contrast, we predict rather than measure and focus on the time to fix a bug after it has been reported, not on the bug’s total lifetime in the code.

Antoniol et al. acknowledge the importance of bug tracking systems and source repositories and suggest a unified framework to manage information extracted from source code, version histories and bug reports [1]. In such a setting a much richer set of features would be available to us, such as links between bug reports and the complexity of the associated source code and change histories for files and components.

Fisher et al. analyze the proximity of software features based on modification and problem report data related to a system’s evolution history [7]. Their approach uncovers relationships between features via bug report analysis and presents them in user-friendly manner, and was also evaluated on Mozilla and Bugzilla. More recently, D’Ambros and Lanza proposed a novel visual approach to finding relationships between software bugs and software evolution [6]. We have focused on predicting bug report resolutions based on bug report data and our work would benefit from a visualization framework.

Weiß et al. employ a text categorization approach to find bug reports that are textually similar to incoming bug reports. They use a nearest neighbors technique to predict how expensive a bug report will be to resolve [17]. To make this prediction, their algorithm uses pre-recorded development cost data from the existing bug report database. This approach is distinct from ours in that it focuses on using the existing database and searching it for similarities, while our approach is intentionally restricted to surface features. Future work in this area might try to assess bug report quality based on textual categorization and a cost measure similar to our “time to resolve” heuristic. Finally, since their cost data did not include the cost of missing a bug report, we were unable to use it in Section 4.4.2.

7. FUTURE WORK AND CONCLUSION

We present a basic linear regression model that predicts whether bug reports are resolved within a give time span. This model is based on bug report features that can be readily extracted from a bug report within a day after its initial filing. The model obtains slightly better F_1 -score performance, which incorporates both precision and recall, than the current practice of triaging all bugs. We show that using more than a day’s worth of data does not significantly improve the model’s performance.

A more detailed analysis of the model demonstrates that the “early features” are the largest contributors to the model’s performance, even when much more data is available. This analysis also showed that self-reported severity is an important factor in the model’s performance. Severity changes over time are not. This is interesting, because severity may not be a reliable indicator of a bug’s importance. Our em-

pirical evaluation of this model shows that it could reduce software maintenance costs if the average cost of triaging a bug report is greater than 2% of the cost of ignoring an important issue.

Future work in this area could explore a variety of different issues. Our experiments only cover the bug reports from a single project. An immediate future step is to extend this work to multiple software projects and then compare how well the trained models generalize. One evaluation might involve testing a model learned on project A as a predictor for project B. Another might involve comparing the learned coefficients from one project with the learned coefficients from another.

Additional future work should address the model itself. First, a more sophisticated model are likely yield better performance than the linear least-squares regression used here. Adding textual categorization techniques, for example, could improve performance significantly. Second, we only explored using this model as a classifier, even though it is capable of outputting continuous values and confidence intervals. This means that it might be possible to use our model as a prioritization tool for triagers rather than just a filter. Such a tool could reduce triaging costs by allowing triagers to work faster. In addition, follow-up work could evaluate the effect of such a tool on triage practices and on the model itself. For example, applying the proposed model as a filter may affect its subsequent prediction quality.

8. ACKNOWLEDGMENTS

We are indebted to Dave Miller for making the Mozilla bug database available to us. We also thank Samuel Sidler for insightful discussions about the bug report triage process.

9. REFERENCES

- [1] G. Antoniol, M. D. Penta, H. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electr. Notes Theor. Comput. Sci.*, 127(3):87–99, 2005.
- [2] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Eclipse ’05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM Press.
- [3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE ’06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM Press.
- [4] R. Bekkerman, R. El-Yaniv, N. Tishby, and Y. Winter. Distributional word clusters vs. words for text categorization. *J. Mach. Learn. Res.*, 3:1183–1208, 2003.
- [5] B. Boehm and V. Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.
- [6] M. D’Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *CSMR*, pages 229–238. IEEE Computer Society, 2006.
- [7] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *WCRE ’03: Proceedings of the 10th Working*

- Conference on Reverse Engineering*, page 90, Washington, DC, USA, 2003. IEEE Computer Society.
- [8] S. Kim and J. E. James Whitehead. How long did it take to fix bugs? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174, New York, NY, USA, 2006. ACM Press.
 - [9] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.
 - [10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM Press.
 - [11] D. R. McCallum and J. L. Peterson. Computer-based readability indexes. In *ACM '82: Proceedings of the ACM '82 conference*, pages 44–48, New York, NY, USA, 1982. ACM Press.
 - [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 *Proceedings of the Open Source software Development Workshop*, pages 155–175, 2002.
 - [15] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.
 - [16] W. Weimer. Patches as better bug reports. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190, New York, NY, USA, 2006. ACM Press.
 - [17] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, May 2007.