Predicting the Severity of a Reported Bug

Ahmed Lamkanfi*, Serge Demeyer*†, Emanuel Giger†, Bart Goethals‡
*LORE - Lab On Reengineering — University of Antwerp, Belgium
†SEAL - Software Evolution and Architecture Lab — University of Zürich, Switzerland
‡ADReM - Advanced Database Research and Modelling — University of Antwerp, Belgium

Abstract—**The severity of a reported bug is a critical factor in deciding how soon it needs to be fixed. Unfortunately, while clear guidelines exist on how to assign the severity of a bug, it remains an inherent manual process left to the person reporting the bug. In this paper we investigate whether we can accurately predict the severity of a reported bug by analyzing its textual description using text mining algorithms. Based on three cases drawn from the open-source community (Mozilla, Eclipse and GNOME), we conclude that given a training set of sufficient size (approximately 500 reports per severity), it is possible to predict the severity with a reasonable accuracy (both precision and recall vary between 0.65-0.75 with Mozilla and Eclipse; 0.70-0.85 in the case of GNOME).**

I. Introduction

During bug triaging, a software development team must decide how soon bugs needs to be fixed, using categories like (P1) as soon as possible; (P2) before the next product release; (P3) may be postponed; (P4) bugs never to be fixed. This so-called *priority* assigned to a reported bug represents how urgent it is from a business perspective that the bug gets fixed. A malfunctioning feature used by many users for instance might be more urgent to fix than a system crash on an obscure platform only used by a tiny fraction of the user base. In addition to the priority, a software development team also keep tracks of the so-called *severity*: the impact the bug has on the successful execution of the software system. While the priority of a bug is a relative assessment depending on the other reported bugs and the time until the next release, severity is an absolute classification. Ideally, different persons reporting the same bug should assign it the same severity. Consequently, software projects typically have clear guidelines on how to assign a severity to a bug. High severity typically represent fatal errors and crashes and low severity typically represent cosmetic issues — depending on the project several intermediate categories exist as well.

Despite their differing objectives, the severity is a critical factor in deciding the priority of a bug. And because the number of reported bugs is usually quite high[1], tool support to aid a development team in verifying the severity of a bug is desirable. Since bug reports typically come with textual descriptions, text mining algorithms are likely candidates for providing such support. Text mining techniques have been previously applied on these descriptions of bug reports to automate the bug triaging process [1, 2, 3] and to detect duplicate bug reports [4, 5]. Our hypothesis is that frequently used terms to describe bugs like "crash" or "failure" serve as good indicators for the severity of a bug. Using a text mining approach, we envision a tool that — after a certain "training period" — provides a second opinion to be used by the development team for verification purposes.

Consequently, in this paper we investigate whether we can accurately predict the severity of a reported bug by analyzing its textual description with a text mining algorithm. While answering this question, we tackle four subsidiary research questions inherent to the use of a text mining algorithm.

- *potential indicators*: Which terms in the textual descriptions of a bug report could serve as good indicators of the severity?
- *short vs. long*: Which (text) fields in the bug reports serve as the best prediction basis? The one-line summary which briefly focusses on the problem or the longer full description which includes more detail?
- *training period*: How many samples must be collected before one can make a reliable predictor?
- *per component vs. cross-component*: Is it better to have a specialized predictor for each component of the software system, or can we combine bug reports over different software components (the so-called "cross-component" approach)?

The paper itself is structured as follows. First, Section II provides the necessary background on both the bug triaging and text mining necessary to understand the technique. The technique and its validation against three open-source cases (Mozilla, Eclipse and GNOME) is then described in Section III. Also concerning validation, we investigate the subsidiary research questions (indicators; short vs. full bug report; the length of the training period; per-component vs. cross-component) in Section IV. After that, Section V lists those issues that may pose a risk to the validity of our results, followed by Section VI discussing related work of other researchers. Finally, Section VII summarizes the results and points out future work.

[1]A software project like Eclipse received over 2.764 bug reports over a period of 3 months (between 01/10/2009-01/01/2010); Mozilla and GNOME received respectively 6.976 and 3.263 reports over the same period.

II. Background

In this section, we provide the necessary background information needed for a detailed description of our approach. First, we discuss the underlying principle of bug reports and bug triaging in general. Then, we provide a brief introduction of the text mining techniques we use in the context of this study.

A. Bug reports and bug triaging

A software bug is what software engineers commonly use to describe the occurrence of a fault in a software system. A fault is then defined as a mistake which causes the software to behave differently from its specifications [6]. Nowadays, users of software systems are encouraged to report the bugs they encounter, using bug tracking systems such as Jira [www.atlassian.com/software/jira] and Bugzilla [www.bugzilla.org].

While reporting a bug, a user is asked to provide information about the bug by filling in a form, used subsequently by the development team to resolve the bug. This form includes a *one-line summary* of the observed malfunction and longer more profound *description*, sometimes including stack traces and the like. Typically the form also allows to select a particular *component* of the faulty software system: e.g., in the Mozilla project, "Bookmarks" and "Layout" are components. For larger software systems the form may even include a field for specifying the *product* where the bug occurred: e.g., in the Mozilla project, the web-browser "Firefox" and the e-mail client "Thunderbird" are products. Most important for the purpose of this paper though is that the user is also asked to make an assessment of the *severity* of the bug. Most projects have clear guidelines on how to determine the severity: Critical = the software will not run; High = unexpected fatal errors (incl. crashes and data corruption); Medium = a feature is malfunctioning; Low = a cosmetic issue. Nevertheless such an assessment must be made in good conscience and when users have no clue they typically just leave the default option.

Researchers have been investigating what characterizes a "good" bug report, i.e., ones that are appreciated by developers and are likely to get fixed sooner [7]. They concluded that "stack traces" and "steps to reproduce" are considered most useful. This is fairly technical information to provide, and unfortunately there is little knowledge on whether users submitting bug reports are capable to do so. Nevertheless, we can make some educated assumptions. Users of technical software such as Eclipse and GNOME typically have more knowledge about software development, hence are more likely to provide the necessary technical detail. Also, a user base which is heavily attached to the software system are more likely to help the developers by writing detailed bug reports.

Finally, caution must be taken as some software systems generate bug reports that are submitted automatically when certain exceptions occur; such reports of course must be omitted as their severity is confirmed by their construction.

B. Text mining techniques

Document classification is widely studied in Machine Learning [8]. Classification or categorization is the process of automatically assigning a predefined category to a document like categorizing textual documents according to their topic. For example, the popular news site Google News [news.google.com] uses a classifier to sort online news reports according to their topic like *entertainment*, *sports* and others. Formally, a classifier is a function

$$f : Document \mapsto \{c_1, ..., c_q\}$$

mapping a document (in our case a bug report) to a certain category in $\{c_1, ..., c_q\}$ (in our case the categories $\{non\text{–}severe, severe\}$. Each document is represented using a vector of features where a feature corresponds to a single term.

A range of classification algorithms exist: like Support Vector Machines, Decision trees, Nearest Neighbor classifier, ... In this study, we use a *Naïve Bayes classifier* which is based on the probabilistic occurrence of terms (the features). This classifier has found its way into many applications including e-mail filtering software where it is used to distinguishes spam from legitimate e-mails based on the contents of the e-mail to some extent. In simple terms, the Naïve Bayesian classifier categorizes documents based on the probability of the presence or absence of a term in the document. When training the classifier, the algorithm keeps track of the probability of each term belonging to a certain category. Using this extracted information, a new document is categorized according to the determined probabilities of each term occurring in the document.

Classifiers based on the Naïve Bayes approach are studied frequently. Even though the Naïve Bayes classification algorithm is based on a simple probabilistic principle, this classifier has proven to perform quite good compared to more sophisticated algorithms [9]. They are called "naïve" because the algorithm assumes that all terms occur independent from each other which is often obviously false [8].

To allow automatic classification of documents, a Naïve Bayes classifiers requires a feature vector, obtained through the following preprocessing steps. The effect of each preprocessing step is shown in Table I.

Table I
Effects of each preprocessing step

Original description	`crashes when I Manage Bookmarks with a Personal Toolbar Folder link`
After stop-words removal	`crashes manage bookmarks personal toolbar folder link`
After stemming	`crash manag bookmark person toolbar folder link`

- *Tokenization:* The process of tokenization consists of dividing a large textual string into a set of tokens where a single token corresponds to a single term. This step also includes filtering out all meaningless symbols like punctuations and commas, because these symbols do not contribute to the classification task. Also, all capitalized characters are replaced by their lower-cased ones.
- *Stop-words removal:* Human languages commonly make use of constructive terms like conjunctions, adverbs, prepositions and other language structures to build up sentences. Terms like "the", "in" and "that" also known as *stop-words* do not carry much specific information in the context of a bug report. Moreover, these terms appear frequently in the descriptions of the bug reports and thus increase the dimensionality of the data which in turn could decrease the performance of classification algorithms. This is sometimes also referred as the *curse of dimensionality*. Therefore, all stop-words are removed from the set of tokens based on a list of known stop-words.
- *Stemming:* The stemming step aims at reducing each term appearing in the descriptions into its basic form. Each single term can be expressed in different forms but still carry the same specific information. For example, the terms "computerized", "computerize" and "computation" all share the same morphological base: "computer". A stemming algorithm like the *porter stemmer* [10] transforms each term to its basic form.

III. Case study

In this section, we first provide a step-by-step in depth description of our approach. Then, we select the measures we use to validate the overall performance of the presented approach. Afterwards, we motivate the selection of the cases and then present the results.

A. Approach

In this study, our approach is based on the assumption that the reporter of a bug uses potentially significant terms in the descriptions which distinguish *non-severe* from *severe* bugs. For example, if it is explicitly stated that the application crashes when performing a certain operation, the hypothesis is that we are most likely dealing with a *severe* bug.

The bug reports we studied originated from Bugzilla bug tracking systems where the severity varies from *trivial*, *minor*, *normal*, *major*, *critical* to *blocker*. There exist clear guidelines on how to assign the severity of a bug. Bugzilla also allow users to request features using the reporting mechanism in the form of a report with "severity" *enhancement*. These reports are not considered in this study since they technically do not represent real bug reports. In our approach, we treat the severities *trivial* and *minor* as *non-severe*, while reports with severity *major, critical, blocker* are considered *severe* bugs. Herraiz et al. proposed a similar grouping of severities [11]. In our case, the *normal* severity is deliberately not taken into account. First of all because they represent the grey zone, hence might confuse the classifier. But more importantly, because in the cases we investigated this "normal" severity was the default option for selecting the severity when reporting a bug and we suspected that many reporters just did not bother to consciously asses the bug severity. Manual sampling of bug reports confirmed this suspicion.

Of course, the prediction must be based on problem-domain specific assumptions. In this case, the predicted severity of a new report is based on characteristics observed in previous ones. Therefore we use a prediction heuristic which learns the specific characteristics of bug reports from a history of bug reports we provide where the severity of each report is known in advance. Subsequently, the heuristic can then be deployed to predict the severity of a previously unseen report. The provided history of bug reports is also known as the *training set* of bug reports. A separate *evaluation set* of reports is used to evaluate the accuracy of the prediction heuristic.

The approach presented in this paper basically consists of the following five steps, detailed below.

(1) Extract and organize bug reports: To have good predictors for the severity of a bug report, the terms used to describe bugs are most likely specialized for the part of the system they are reporting about. Bug reports are typically organized according to the affected *component* and the corresponding *product*. The first step of our approach consequently selects bug reports of a certain product and component.

(2) Preprocessing the bug reports: To assure the optimal performance of the text mining algorithm, we apply the standard preprocessing steps for textual data (tokenization, stop-words removal and stemming) on the descriptions in the bug reports.

(3) Choosing a training and evaluation set: As is common in text classification, we train the heuristic by giving a set of example bug reports where their severities is known in advance. The training set with the example reports is selected from the global set of the bug reports in a random manner. To ensure that the classifier is not affected by the distribution of the bugs according to their severities, we make sure that we select just as much reports in the training and evaluation set for each severity.

We expect the size of the training set to play a significant role for the prediction heuristic. We use $\pm 70 - 30\%$ of the available reports for the training and evaluation set respectively. However, we will investigate this issue further under the subsidiary research questions in IV-C.

(4) Training the classifier: Using the training set we obtained in the previous step, we now advance to the actual training period where the Naïve Bayes classification

algorithm basically learns the characteristics of the bug reports.

(5) Applying the classifier on the evaluation set: Once the classifier is trained sufficiently, we apply it on the bug reports contained in the evaluation set where the heuristic predicts the severity of each report. Since the severity for these bug reports is known in advance, we can compare the predictions to the actual severities in order to verify how accurate the predictions would have been.

B. Evaluation measures

The two most commonly used evaluation metrics which we use to validate our approach are *precision* and *recall*.

- ***Precision*** The percentage of bug reports predicted as either *non-severe* or *severe* which are correctly predicted. We consider precision thus for each severity separately. When S is either *non-severe* or *severe*, we define precision more formally as:

$$Precision_S = \frac{\textit{\# bugs correctly predicted as S}}{\textit{\# bugs predicted as S}}$$

- ***Recall*** The percentage of all bug reports with severity *non-severe* or *severe* that are actually predicted as being respectively *non-severe* or *severe*. Here, we also consider recall for each precision separately. When S is either *non-severe* or *severe*, we define recall more formally as:

$$Recall_S = \frac{\textit{\# bugs correctly predicted as S}}{\textit{\# bugs of severity S}}$$

We calculate both precision and recall using a *confusion matrix*, sketched in Table II. This matrix represents all possible outcomes when making predictions of the severity.

Table II
CONFUSION MATRIX USED TO CALCULATE PRECISION AND RECALL

		Correct severity	
		non-severe	*severe*
Predicted severity	*non-severe*	*tp: true positives*	*fp: false positives*
	severe	*fn: false negatives*	*tn: true negatives*

Using this confusion matrix, we calculate precision and recall as follows:

$$prec_{non-severe} = \frac{tp}{tp + fp} \qquad prec_{severe} = \frac{tn}{tn + fn}$$

$$rec_{non-severe} = \frac{tp}{tp + fn} \qquad rec_{severe} = \frac{tn}{tn + fp}$$

With an ideal classifier, all bug reports are classified correctly (high recall), while it minimizes the number of incorrect classified bug reports (high precision).

We also use an alternate technique to evaluate the performance of our approach: the Receiver Operating Characteristic (ROC). The ROC compares the rate of true positives (TPR) with the rate of false positives (FPR) and is typically drawn as a curve [12]. The "area under the ROC curve" (AUC) is then a statistic summarizing the ROC curve in a single number representing the overall performance of the heuristic. This statistic represents the probability that the outcome of the heuristic is a better indication when compared to randomly choosing the severity. Random classification has an AUC value of 0.5 while the perfect heuristic has an AUC of 1 — similar to precision and recall — which means that the heuristic predicted the severities of all bugs correctly. Therefore, the higher AUC value is, the better the heuristic performs.

C. Selection on the cases

To validate the presented approach, we use bug reports from three major open-source projects using Bugzilla as their bug tracking system: Mozilla, Eclipse and GNOME.

Mozilla: [http://bugzilla.mozilla.org] Mozilla is an open-source software project hosting several popular products like *Firefox* and *Thunderbird*. The copy of the bug databases we obtained contains all reports submitted in the period of 1997-2008 corresponding to approximately 400.000 reported bugs. The Mozilla products are examples of applications with less savvy users so we expect the bug reports to be less detailed than the ones from Eclipse and GNOME.

Eclipse: [http://bugs.eclipse.org/bugs] Eclipse is an open-source integrated development environment widely used in both open-source and industrial settings. The bug database contains over 200.000 bug reports submitted in the period of 2001-2008. Eclipse is a technical application used by developers themselves, so we expect the bug reports to be quite detailed and "good" (as defined by Bettenburg et al. [7]).

GNOME: [http://bugzilla.gnome.org] GNOME is an open-source desktop-environment developed for Unix-based operating systems. In this case we have over 450.000 reported bugs available submitted in the period of 1998-2009. GNOME was selected primarily because it was part of the MSR 2010 mining challenge [msr.uwaterloo.ca/msr2010/challenge/]. As such the community agreed that this is a worthwhile case to investigate. Moreover results we obtained here might be compared against results obtained by other researchers. For our investigation, GNOME was also interesting because —as confirmed by our inspection of the bug reports— it had a lot of automatically generated bug reports.

D. Selection of the components

As mentioned under the first step of our approach, the predictive heuristic is considered per component so that "component-specific" terms in the bug reports are taken into account by the heuristic. Therefore, we select a series of components from the cases on which we apply and validate our approach. We selected components according to the highest number of available bug reports since these contain the most reports to train and validate with. In Table III, we present the selected components with their corresponding identification ID. The ID's of the GNOME components are omitted because this information was not included in the available data we used.

Table III
COMPONENTS AND NUMBER OF THEIR (*non-*)*severe* BUGS FOR RESPECTIVELY MOZILLA, ECLIPSE AND GNOME

ID	Name	***Non-severe* bugs**	***Severe* bugs**
23	Layout	1.043	3.064
145	Bookmarks	644	1.120
279	FirefoxGeneral	2.699	8.443
8	Eclipse User Interface	1.403	3.258
12	JDT User Interface	1.401	1.522
43	JDT Text	806	557
x	Calendar	619	2.661
x	Contacts	638	1.637
x	Mailer	2.537	7.239

E. Results

Table IV shows the precision and recall measures for each of the selected components from Mozilla, Eclipse and GNOME.

Table IV
PRECISION AND RECALL OF THE APPROACH APPLIED ON A SERIES OF COMPONENTS

	Non-severe		***Severe***	
Component	Precision	Recall	Precision	Recall
Mozilla:Layout	0.701	0.785	0.752	0.653
Mozilla:Bookmarks	0.692	0.703	0.698	0.687
Mozilla:FirefoxGeneral	0.692	0.744	0.723	0.670
Eclipse:UI	0.707	0.633	0.668	0.738
Eclipse:JDT-UI	0.653	0.714	0.685	0.621
Eclipse:JDT-Text	0.705	0.718	0.713	0.700
GNOME:Calendar	0.828	0.783	0.794	0.837
GNOME:Contacts	0.767	0.706	0.728	0.785
GNOME:Mailer	0.767	0.804	0.794	0.756

As we see in Table IV, the results of the Mozilla and Eclipse components are similar where we note that both precision and recall vary between the values 0.65-0.75. This applies for both *non-severe* as *severe* bugs. Moreover, the GNOME case shows significantly better results with precision and recall varying in the range 0.70-0.85.

Table V shows the AUC values of the predictive heuristic applied on the three cases. The AUC for all Eclipse components are approximately 0.74. We notice an improvement with the Mozilla components where we observe an AUC of approximately 0.80. The approach performs best with the components of GNOME where the AUC varies between 0.82-0.87. In this case, this means that our approach performs around 35 % better than if we would randomly guess the severity of each bug.

Table V
AUC MEASURES OF THE APPROACH ON A SERIES OF COMPONENTS

Component	**AUC**
Mozilla:Layout	0.813
Mozilla:Bookmarks	0.793
Mozilla:FirefoxGeneral	0.802
Eclipse:UI	0.744
Eclipse:JDT-UI	0.740
Eclipse:JDT-Text	0.775
GNOME:Calendar	0.869
GNOME:Contacts	0.822
GNOME:Mailer	0.854

Therefore, we conclude that it is possible to predict the severity of a reported bug based on the provided information, more particularly the one-line summary using a Naïve Bayes classifier. The accuracy of the approach is reasonable, yet it depends on the case.

IV. SUBSIDIARY RESEARCH QUESTIONS

Knowing that we can we can accurately predict the severity of a reported bug, there are some issues which may affect the performance of the technique. We formulate these issues using the following questions:

- Which terms in the textual descriptions of a bug report could serve as good indicators of the severity?
- A bug reporter also provides a long description in a report. How does the prediction perform when training with this description?
- How many training examples do we need to build a stable, robust predictor?
- What about training the heuristic with *cross-component* bug reports?

We make variations to some of the parameters of the approach and report the differences compared to the base results shown in Table IV.

A. Which terms in the textual descriptions of a bug report could serve as good indicators of the severity?

The Naïve Bayes classifier we use in our approach predicts the severity based on the probability of the presence or absence of a term in the one-line summary of a report. When we train this classifier, the algorithm calculates these probability values of each term belonging to a certain severity. We managed to extract these values from the classifier of each term. Based on these extracted probabilities, we report for a number of components in Table VI the top-10 most significant terms indicating the severity.

From Table VI we observe that terms like `deadlock`, `hang` and `segfault` are significant terms considering *severe* reported bugs. This confirms our hypothesis that certain terms in the descriptions are good indicators for the severity of a reported bug. The same terms tend even to appear across different components and even products.

Table VI
TOP-10 MOST SIGNIFICANT TERMS INDICATING EACH SEVERITY

Component	*Non-severe*	*Severe*
Mozilla Firefox-General	`inconsist, favicon, credit, extra, consum, licens, underlin, typo, inspector, titlebar`	`fault, machin, reboot, reinstal, lockup, seemingli, perman, instantli, segfault, compil`
Eclipse JDT UI	`deprec, style, runnabl, system, cce, tvt35, whitespac, node, put, param`	`hang, freez, deadlock, thread, slow, anymor, memori, tick, jvm, adapt`
GNOME Mailer	`mnemon, outbox, typo, pad, follow, titl, high, acceler, decod, reflect`	`deadlock, sigsegv, relat, caus, snapshot, segment, core, unexpectedli, build, loop`

When considering the *non-severe* indicators, we observed that some terms like `typo` serve as good indicators. However, we noticed that less *non-severe* indicators are shared across different components while *severe* indicators tend to appear across different components and products. This can be explained by the origins of *severe* bugs which are typically easier to describe using specific terms. For example, the application crashes or there is a memory issue. These situations are easily described using specific powerful terms like `crash` or `memory`. This is less obvious in the case of *non-severe* indicators since they typically describe cosmetic issues. In this case, reporters use less common terms to describe the nature of the problem.

B. How does the approach perform with the longer description?

In addition to the one-line summary, the reporter also provides a longer description of the encountered problem when a bug is submitted. This description provides more detailed information of the bug. For example, the reporter explains when exactly the bug occurs and how it can be reproduced. The reporter may also provide a stack trace of the application when the bug occurs within this description. This information may also implicitly contain significant information in the context of the severity of the bug. Therefore, we now use this description (instead of one-line summary) to train the heuristic expecting that the accuracy of the predictions will improve since the heuristic is provided with more detailed information. For this part of the evaluation we had to exclude the bug reports for the GNOME case since manual inspection revealed that the severity of the reported bug was automatically included into the description field. This will of course jeopardize our results as we would quickly obtain a perfect classifier. Table VII shows the effect on the precision and recall for Mozilla and Eclipse components.

Table VII
PERFORMANCE OF THE APPROACH WHEN USING THE LONGER DESCRIPTION

	Non-severe		*Severe*	
Component	Precision	Recall	Precision	Recall
Mozilla:Layout	0.583	0.961	0.890	0.314
Mozilla:Bookmarks	0.536	0.963	0.820	0.166
Mozilla:FirefoxGeneral	0.578	0.948	0.856	0.308
Eclipse:UI	0.548	0.976	0.892	0.197
Eclipse:JDT-UI	0.547	0.973	0.881	0.195
Eclipse:JDT-Text	0.570	0.988	0.955	0.257

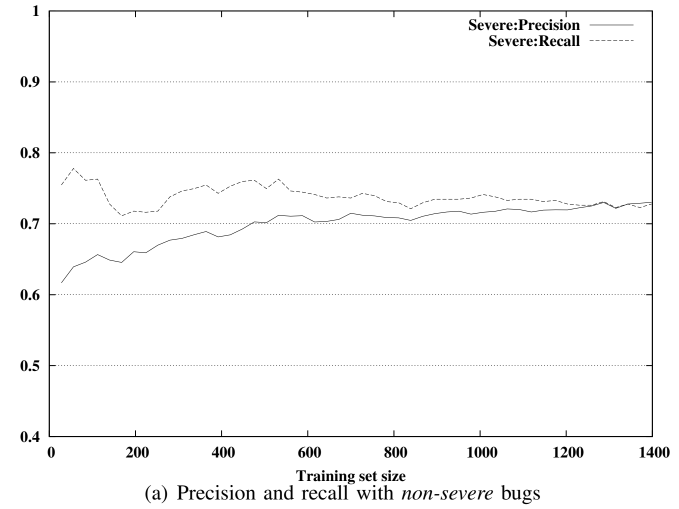
The first striking observation we make from the results in Table VII, is the contrast with the results of our approach. While we see on the one hand very high recall and much lower precision values with *non-severe*, we also observe on the other hand a very high precision and a much lower recall with *severe*. In the context of the confusion matrix (see Table II) it seems we are dealing with large number of false positives and a very small number of false negatives. This basically means that many *severe* bug reports are faulty predicted as *non-severe* explaining the low recall value in the *severe* case. This subsequently leads to a small number of remaining *severe* reported bugs where the heuristic predicts correctly (leading to a high precision). Also, a small number of *non-severe* reports are correctly predicted, but due to the large number of false positives we conclude that these predictions are not meaningful. Therefore, we conclude that the one-line summary of the bug is a better source of information than the longer full description for the heuristic.

The contrast of the obtained results can be explained by the characteristics of the description and confirms earlier results from a linguistic analysis of bug reports [13]. The description generally tends to be much longer in size compared to the one-line summary. The reporter explains the nature of the problem using several sentences while the same information is also summarized using only one line. The information is often scattered in a longer description and therefore more difficult to extract resulting in a degradation of the performance. In some cases the reporter also includes stack traces, snippets of source code in these descriptions which is considered as noise by the heuristic.

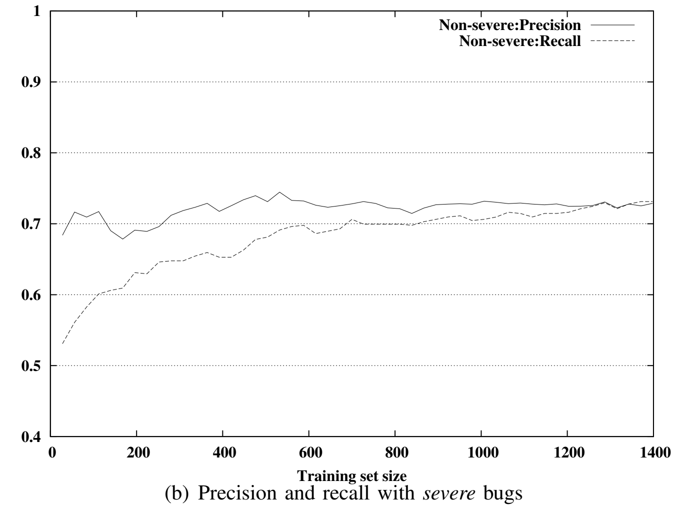
C. How many training examples?

Accepting that we can train an algorithm to predict the severity, we of course would like to know how many bug reports we need in order to obtain good, stable predictions. Therefore we ran a series of measurements, where we gradually increase the size of the training set. However, we maintain only one single fixed evaluation set which we repeatedly use to evaluate the performance with each increment of the training set. Figure 1, 2 (a) and (b) show the performance of the heuristic when we vary the training set sizes for respectively the Mozilla *FirefoxGeneral* and

Figure 1. Performance of the classifier in function of the training-set size with the Mozilla *GeneralFirefox* component

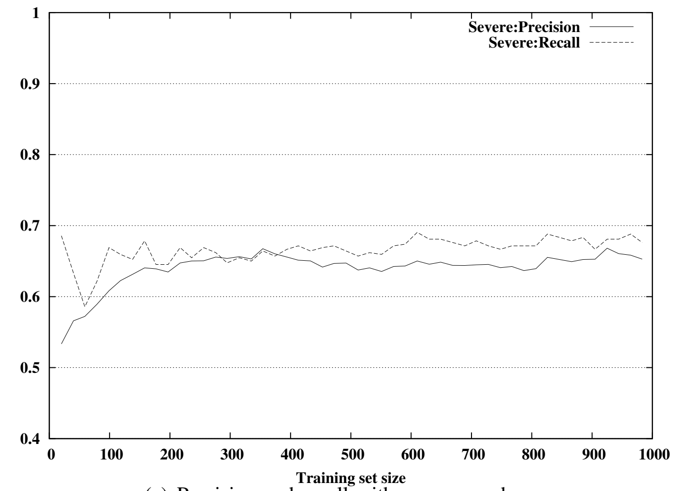


(a) Precision and recall with *non-severe* bugs

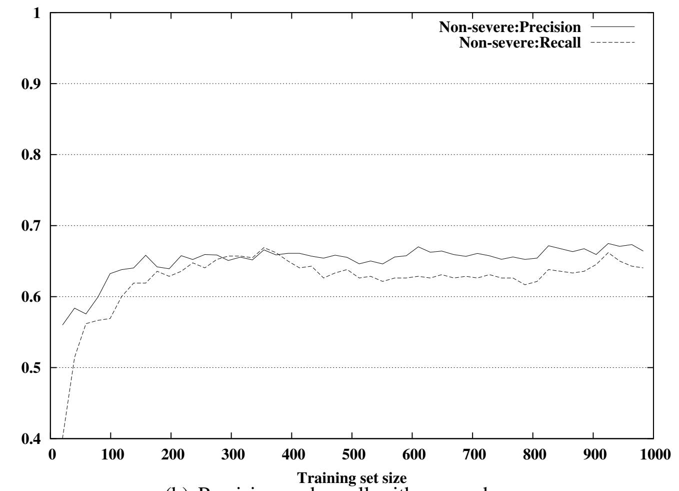


(b) Precision and recall with *severe* bugs

Figure 2. Performance of the classifier in function of the training-set size with the Eclipse *User Interface* component



(a) Precision and recall with *non-severe* bugs



(b) Precision and recall with *severe* bugs

Eclipse *User Interface* components. Due to space limitations we only show the results for the two components with the most reports in Mozilla and Eclipse case. We observed similar trends for all other components in the the three cases however. Note that with the Mozilla case, the training set increases up to 1400 reports while in the Eclipse case, we only have a maximum training set with over 1000 reports.

Figure 1 and 2 show the accuracy of the predictions when we use training sets of different sizes. Along the x-axis, we see how many training examples we have used in each case which we relate to precision and recall measures we obtained using a fixed evaluation set. Both figures clearly show a very unstable behavior when a relatively small number of training examples are used. The impact of small increments in the training set seem to have a significant impact on the performance. This indicates that the heuristic has not yet "learned" sufficient about the specifics of bug reports and therefore the heuristic is easily influenced by the increment (which is large relatively to the number of training examples in the previous run).

However, the performance of our approach stabilizes in both the Mozilla, Eclipse and GNOME components when the heuristic is trained with more bug reports. With approximately more than 500 bug reports for each severity, both precision and recall stabilize and we conclude that from then onwards the heuristic has learned sufficiently to allow for more reliable predictions.

D. *What about a cross-component approach?*

In our approach, we train and evaluate our heuristic separately for each component. However, we were also interested to know whether this specialization is really necessary. In other words, can the information stored in bug reports of one component be used to predict the severity of reports from another component? In a new measurement, we combine bug reports from different components in a single training and evaluation set. Table VIII shows the accuracy of this modified approach.

Table VIII
PERFORMANCE OF THE CROSS-COMPONENT APPROACH

	Non-severe		***Severe***	
Runs	Precision	Recall	Precision	Recall
Mozilla: ± 500	0.673	0.620	0.648	0.700
Mozilla: ± 1.000	0.702	0.742	0.726	0.685
Mozilla: ± 2.000	0.704	0.750	0.733	0.685
Eclipse: ± 500	0.672	0.503	0.603	0.754
Eclipse: ± 1.000	0.673	0.510	0.605	0.752
Eclipse: ± 2.000	0.693	0.553	0.628	0.755
GNOME: ± 500	0.753	0.680	0.708	0.777
GNOME: ± 1.000	0.822	0.727	0.755	0.842
GNOME: ± 2.000	0.817	0.737	0.760	0.835

As shown in Table VIII, we performed the cross-component approach for each case where we also varied the size of the training set (e.g. training set varies from 500 to 2000 training reports per severity). The new training set is constructed from training examples which we randomly selected from five different components, including the ones we presented in the previous evaluations.

From the results of the Mozilla case, we see an improvement in performance when we use a training set with 1000 reports for each severity compared to when using 500 reports. In the Mozilla case, the performance is similar with our previous approach: precision and recall vary between 0.65-0.75. However, the performance is lower when we only use for each severity around 500 training examples. Therefore, we need a larger training set when we perform cross-component predictions here. In the case of Eclipse, we notice that the cross-components predictions are less obvious. Even when the training set contains more than 2000 example reports for each severity, the prediction heuristic still performs poorly. In the GNOME case, we notice an increase in the overall performance when the training set expands.

This phenomenon can be explained by the fact that we are dealing with different cases. Each case has their own specific characteristics when it comes to the one-line summary in the bug report. These problem-specific characteristics could be even more specialized to a single component resulting in a poor performing "cross-component" predictor. However, some cases (Mozilla and GNOME) tend to share problem-specific characteristics across different components and products. In this case we see an improvement of the performance of this approach.

V. Threats to validity

In this section we identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [14, 15]) we organize them in four categories.

Construct Validity: We have trained our classifier per component, assuming that special terminology used per component will result in a better prediction. However, bug reporters have confirmed that providing the "component" field in a bug report is notoriously difficult [7], hence we risk that the users interpreted these categories in different ways than intended. We alleviated the risk by selecting those components where there are many bug reports (see Table III). We verified that few of these bugs have been reassigned to other components.

Internal Validity: Our approach relies heavily on the presence of a causal relationship between the contents of the fields in the bug report and the severity of the bug. There is empirical evidence that this causal relationship indeed holds (see for instance [13]). Nevertheless software developers and bug reporters confirmed that other fields in the bug report or more important, which may be a confounding factor [7].

External Validity: In this study, we focused on the bug reports of three software projects: Mozilla, Eclipse and GNOME. Like in other empirical studies, the results obtained from our presented approach are therefore not guaranteed to hold with other software projects. However, we selected the cases to represent worthwhile points in the universe of software projects, representing sufficiently different characteristics to warrant comparison. For instance, Eclipse was selected because its user base are developers hence likely to produce "good" bug reports. This was in contrast with Mozilla and to some extent GNOME.

The bug reports used in our approach are extracted from cases using Bugzilla as their bug tracking system. Other bug tracking systems exist as well like Jira and CollabNet. Since they potentially use other representations of bug reports, it may be possible that the approach must be adapted to the context of other bug tracking systems.

Reliability: Since we use the bug reports submitted by the community both as training and evaluation purposes, it is not guaranteed that the severities in these reports are entered correctly. Users fill in the reports according to their understanding and therefore assess severities corresponding to their experience, which do not necessarily correspond with the guidelines. We explicitly omitted the bug reported with severity "normal" since this category corresponded to the default option when submitting a bug and thus likely to be unreliable. We performed a manual inspection of a sample of the bug reports to verify whether the severity was properly accounted for and found it OK in almost all of the cases.

The tools we used to process the data might contain errors. We implemented our approach in the programming language *Ruby* [*www.ruby-lang.org*] where we relied as much as possible on standard tools and libraries for the querying of the data (i.e., REXML which is the ruby standard library for XML parsing) and the Bayes classifier (i.e., the Ruby Porter stemmer [*rubyforge.org/projects/stemmer*] and the Ruby Classifier [*classifier.rubyforge.org*] library) and use a small amount of scripting to glue everything together. Hence we believe this risk to be acceptable.

VI. Related work

At the moment, we are only aware of a single work on the automatic prediction of the severity of reported bugs. Menzies et al. predict the severity based on a rule learning technique which also uses the textual descriptions of reported bugs [16]. The approach was applied on five projects supplied by the NASA's Independent Verification and Validation Facility. In this case-study, the authors have shown that it is feasible to predict the severity of bug reports using a text mining technique even for a more fine-grained categorization than we do (the paper distinguishes between 5 severity levels of which 4 were included in the paper). While they were forced to use smaller training sets than we do (the data sets sizes ranged from 1 to 617 bug reports per severity), the precision and recall they reported varied a lot more (precision between 0.08 and 0.91; recall between 0.59 and 1.00). This suggests that the training sets indeed must be sufficiently large to arrive at stable results. Due to the smaller cases, Menzies et. al. could also not investigate the subsidiary research questions we reported about in section IV

Antoniol et al. also used text mining techniques on the descriptions of reported bugs to predict whether a report is either a real bug or a request for an enhancement [17]. They used techniques like decision trees, logistic regression and also a Naïve Bayesian classifier for this purpose. The performance of this approach on three cases (Mozilla, Eclipse and JBoss) indicated that reports can be predicted to be a bug or an enhancement with between 77% and 82% correct decisions.

Other current research concerning bug characterization and prediction mainly apply text mining techniques on the descriptions of bug reports. This work can be divided in two groups: automatically assigning newly reported bugs to an appropriate developer based on his or her expertise and detecting duplicate bug reports.

A. Automatic bug assignment

Machine learning techniques are used to predict the most appropriate developer for resolving a new incoming bug report. This way, bug triagers are assisted in their task. Cubranic et al. trained a Naïve Bayes classifier with the history of the developers who solved the bugs as the category and the corresponding descriptions of the bug reports as the data [3]. This classifier is subsequently used to predict the most appropriate developer for a newly reported bug. Over 30 % of the incoming bug reports of the Eclipse project are assigned to a correct developer using this approach.

Anvik et al. continued investigating the topic of the previous work and performed new experiments in the context of automatic bug assignment. The new experiment introduced more extensive preprocessing on the data, introducing more classification algorithms like Support Vector Machines. In this case, an overall classification accuracy of 57 % and 64 % for the Eclipse and Firefox projects respectively [1].

B. Duplicate bug report detection

Since the community behind a project is in some cases very large, it is possible for multiple users to report the same bug into the bug tracking system. This leads to multiple bug reports describing the same bug. These "duplicate" bug reports result in more triaging work. Runeson et al. used text similarity techniques to help automate the detection of duplicate bug reports by comparing the similarities between bug reports [4]. In this instance, the description was used to calculate the similarity between bug reports. Using this approach, over 40 % of the duplicate bug reports are correctly detected.

Wang et al. consider not only the actual bug reports, but also include "execution information" of a program which is for example the execution traces [5]. This additional information reflects the situation that lead to the bug and therefore reveal buggy runs. Adding structured and unambiguous information to the bug reports and comparing it to others, improves the overall performance of the duplicate bug report detection technique.

VII. Conclusions and future work

Deciding how soon a reported bug needs to be fixed partly depends on its severity. However, estimating the severity of a bug is often left to the person reporting the bug. This paper shows that it is possible to predict the severity based on other information contained in a bug report, in particular the textual information describing the bug. We evaluated the performance of predictions based on three cases drawn from the open-source community (Mozilla, Eclipse and GNOME). We conclude that it is possible to predict the severity of a reported bug given sufficient training data ($\pm$ 500 bug reports for each severity) with a reasonable performance where both precision and recall vary between 0.65-0.75 for selected components of Mozilla and Eclipse. In the case of the GNOME components, we have seen a notable increase in performance with both precision and recall varying between 0.70-0.85.

Knowing that we can accurately predict the severity of a reported bug, we investigated whether certain parameters affect the accuracy of the result. In particular, we had a look at the terms used in the textual descriptions of a bug report and discovered that they could serve as indicators but were largely project and component dependent. This specialization is indeed key: we also ran a test to see the effect of cross-component predictors and observed a weaker performance. We also investigated whether the longer description included in a bug report would result in a better predictor, but for the systems we investigated this was not the case.

This study is relevant because it enables us to implement a more automated and more efficient bug triaging process. It also can contribute to the current research regarding bug triaging. We see trends concentrating on automating the triaging process where this current research can be combined with our approach with the intention to improve the overall reliability of a more automated triaging process.

Future work is aimed at including additional sources of data to support our predictions. Information from the (longer) description will be more thoroughly preprocessed so that it can be used for the predictions. In our approach, we used the Naïve Bayes based classification algorithm. However, other classification algorithms also exist including the promising *Decision trees* and *Support Vector Machines* could be used for our purpose. The performance of each of these classifications will be compared in order to determine the most efficient approach for our goal. Also, the impact of additional techniques like feature selection and cross-validation will be evaluated. Finally, we will investigate other industrial cases, where fewer bug reports get submitted but where the bug reports get reviewed consciously.

Acknowledgments

We would like to thank Andre Klapper who is an experienced bug triager for the interesting discussions we had during this study. We would also like to thank Sandro Boccuzzo, Giacomo Ghezzi and Michael Würsch [SEAL – University of Zürich], Andy Zaidman [SWERL – TU Delft], Joris Van Geet [LORE – University of Antwerp] for reviewing earlier drafts of this paper.

This work has been carried out in the context of a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Additional sponsoring by (i) the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*; (ii) the Research Foundation – Flanders (FWO) sponsoring a sabbatical leave of Prof. Serge Demeyer.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006.

[2] J. Gaeul, K. Sunghun, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the European Software Engineering Conference 2009*. ACM, 2009, pp. 111–120.

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

[4] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering*, 2007.

[5] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, 2008.

[6] R. Patton, *Software Testing (2nd Edition)*. Sams, 2005.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[8] R. Feldman and J. Sanger, *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, December 2006.

[9] I. Rish, "An empirical study of the naive bayes classifier," in *Workshop on Empirical Methods in AI*.

[10] M. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[11] I. Herraiz, D. German, J. Gonzalez-Barahona, and G. Robles, "Towards a Simplification of the Bug Report Form in Eclipse," in *5th International Working Conference on Mining Software Repositories*, May 2008.

[12] D. M. Green and J. A. Swets, *Signal Detection Theory and Psychophysics*. John Wiley and Sons, Inc., 1966.

[13] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, 2006, pp. 127–134.

[14] R. K. Yin, *Case Study Research: Design and Methods, 3 edition*. Sage Publications, 2002.

[15] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, 2009.

[16] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *IEEE International Conference on Software Maintenance*, 28 2008-Oct. 4 2008, pp. 346–355.

[17] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *CASCON '08: Proceedings of the conference of the center for advanced studies on collaborative research*. ACM, 2008, pp. 304–318.