Empir Software Eng (2015) 20:1354–1383
DOI 10.1007/s10664-014-9331-y

Automated prediction of bug report priority using multi-factor analysis

Yuan Tian · David Lo · Xin Xia · Chengnian Sun

Published online: 3 August 2014

Abstract Bugs are prevalent. To improve software quality, developers often allow users to report bugs that they found using a bug tracking system such as Bugzilla. Users would specify among other things, a description of the bug, the component that is affected by the bug, and the severity of the bug. Based on this information, bug triagers would then assign a priority level to the reported bug. As resources are limited, bug reports would be investigated based on their priority levels. This priority assignment process however is a manual one. Could we do better? In this paper, we propose an automated approach based on machine learning that would recommend a priority level based on information available in bug reports. Our approach considers multiple factors, `temporal`, `textual`, `author`, `related-report`, `severity`, and `product`, that potentially affect the priority level of a bug report. These factors are extracted as features which are then used to train a discriminative model via a new classification algorithm that handles ordinal class labels and imbalanced data. Experiments on more than a hundred thousands bug reports from Eclipse show that we can outperform baseline approaches in terms of average F-measure by a relative improvement of up to 209 %.

Keywords Bug report management · Priority prediction · Multi-factor analysis

Communicated by: Yann-Gaël Guéhéneuc and Tom Mens

Y. Tian (✉) · D. Lo
School of Information Systems, Singapore Management University, Singapore, Singapore
e-mail: yuan.tian.2012@phdis.smu.edu.sg

D. Lo
e-mail: davidlo@smu.edu.sg

X. Xia
College of Computer Science and Technology, Zhejiang University, Hangzhou, China
e-mail: xinxia@smu.edu.sg

C. Sun
University of California at Davis, Davis, CA 95616 USA
e-mail: chengniansun@gmail.com

1 Introduction

Due to system complexity and inadequate testing, many software systems are often released with defects. To address these defects and improve the next releases, developers need to get feedback on defects that are present in released systems. Thus, they often allow users to report such defects using bug reporting systems such as Bugzilla, Jira, or other proprietary systems. Bug reporting is a standard practice in both open source software development and closed source software development (e.g., Windows).

Developers are often overwhelmed with the large number of bug reports (Anvik et al. 2005). Prioritizing bug reports can help developers manage the bug triaging process better. Developers often leave bug reports unfixed for years due to various factors including time constraints (Xia et al. 2014). Thus, it is important for developers to prioritize bug reports well so that important reports are prioritized and fixed first. Bug report prioritization is especially important for large projects that are used by many clients since they typically receive higher numbers of bug reports.

Prioritizing bugs is a manual process and is time consuming. Bug triagers need to read the information provided by users in the new bug reports, compare them with existing reports, and decide the appropriate priority levels.

To aid bug triagers in assigning priority, we propose a new automated approach to recommend priority levels of bug reports. To do so, we leverage information available in the bug reports. Bug reports contain various information including short and long descriptions of the issues users encounter while using the software system, the products that are affected by the bugs, the dates the bugs are reported, the people that report the bugs, the estimated severity of the bugs, and many more. We would like to leverage this information to predict the priority levels of bug reports.

Our approach predicts the priority level of bug reports by considering several factors that could affect the priority of a bug report. These factors include other bug reports that are reported at the same time as the bug report (`temporal`), the textual content of the bug report (`textual`), the author of the bug report (`author`), related bug reports (`related-report`), the estimated severity of the bug (`severity`), and the product which the reported bug affects (`product`). We extract various features, e.g., the number of bugs reported in the past 7 days, etc., to capture each of these factors.

Next, we propose a new machine learning approach, in particular a new classification algorithm, to create a model from the features that could predict if a bug report should be assigned a priority level P1, P2, P3, P4, or P5. We take a training set of reports along with the priority levels. Feature values are then extracted from each data point (i.e., bug report) in this training set. The machine learning algorithm would then decide the likely priority level of a bug report whose priority level is to be predicted based on these feature values.

We propose a new framework named DRONE to aid triagers in assigning priority labels to bug reports. Inside DRONE, we include our new classification engine named GRAY which *enhances* linear regression with *our thresholding approach* to handle imbalanced bug report data. Linear regression models the relationship between the values of the various features and the priority levels (which takes a value between 1 to 5) of bug reports. Linear regression considers the priority levels as ordinal values (i.e., P1 is closer to P2 than it is to P5) rather than categorical values (i.e., P1 is as different to P2 as it is to P5). It then outputs a real number given a set of values of the different features of a new bug report. Thresholding learns a set of *thresholds* defining a set of *ranges* for the outputs of the linear regression model where each range corresponds to a priority level. Due to the data imbalance, the best set of ranges are of unequal sizes and these ranges could be effectively learned from a set of validation data points, thus addressing the data imbalance issue.

Closest to our work, is the series of work on bug report severity prediction by Menzies and Marcus (2008), and Lamkanfi et al. (2010, 2011), and our own previous work (Tian et al. 2012). These studies predict the severity field of a bug report based on the textual content of the report. Severity however is different from priority. Severity is assigned from a user perspective while priority is assigned based on the developers' perspective. We have checked with an experienced developer from Eclipse Project Management Committee, who has fixed hundreds of bugs in Eclipse. He states that:

> Severity is assigned by customers [users] while priority is provided by developers …customer [user] reported severity does impact the developer when they assign a priority level to a bug report, but it's not the only consideration. For example, it may be a critical issue for a particular reporter that a bug is fixed but it still may not be the right thing for the eclipse team to fix.

Thus our work is different from the work on bug severity prediction. In this work, we predict the priority of bug reports by considering the `temporal`, `textual`, `author`, `related-report`, `severity`, and `product` factors of a bug report. This holistic view of a bug report is needed for us to support triagers in assigning priority levels to bug reports.

We experiment our solution on more than a hundred thousand bug reports of Eclipse that span a period of several years. We compare our approach with a baseline solution that adapt an algorithm by Menzies and Marcus (2008) for bug priority prediction. Our experiments demonstrate that we can achieve up to 209 % improvement on the average F-measure.

The contributions of this work are as follows:

1. We propose a new problem of predicting the priority of a bug given its report. Past studies on bug report analysis has only considered the problem of predicting the severity of bug reports which is an orthogonal problem.
2. We predict priority by considering the different factors that potentially affect the priority level of a bug report. In particular, we consider the following factors: `temporal`, `textual`, `author`, `related-report`, `severity`, and `product`.
3. We propose a new classification engine, named GRAY, which is a component of DRONE, that *enhances* linear regression with *thresholding* to handle imbalanced data.
4. We have experimented our solution on more than a hundred thousands bug reports from Eclipse in its ability to support developers in assigning priority levels to bug reports. The result shows that DRONE could outperform a baseline approach, built by adapting a bug report severity prediction algorithm, in terms of average F-measure, by a relative improvement of up to 209 %.

The structure of this paper is as follows. In Section 2, we describe preliminary information on bug report, text pre-processing, and measuring similarity of bug reports. In Section 3, we describe our proposed approach. We describe our experiments setup in Section 4. Section 5 presents the result of our experiments. Next, we discuss interesting issues in Section 6. Related work is presented in Section 7. Finally, we conclude and discuss future work in Section 8.

2 Preliminaries & Problem Definition

In this section, we first describe bug reports and the bug reporting process. Next, we present an approach to pre-process textual documents. Then, we highlight REP (Sun et al. 2011), which is a recently proposed state-of-the-art similarity measure of bug reports. Finally, we present our problem definition.

2.1 Bug Reports and Reporting Process

Developers often desire feedback on defects that exist in released systems. To collect feedback, bug tracking systems are often employed. Popular bug tracking systems include Bugzilla, Jira, and other proprietary systems. Utilizing these systems, users can report issues that they find in the system and track their progress. Each reported issue is referred to as a bug report.

Each bug report contains information on how the bug could be reproduced plus other related information that could help in debugging. In a bug report, there is information on short and long descriptions of the bug, and various information on the product that is affected by the bug, the component that is affected by the bug, the estimated severity of the bug, the date that the bug is reported, and many more. All this information is commonly provided by bug reporters when they submit bug reports. We provide a description of fields in a bug report that are of interest to us in Table 1.

When a new bug report is submitted into a bug tracking system, a bug triager would first investigate the fields of the bug report and potentially other reports. Based on the investigation, he or she would check the validity of the bug report and may change values of some

Table 1 Fields of interest in a bug report

Field	Description
Summary	Summarized description of a bug. Typically this summary only contains a few keywords
Description	Long description of a bug. Typically this would include information that would help in the debugging process including the reported error message, the steps to reproduce the error, etc.
Product	The product which is affected by the bug
Component	The component which is affected by the bug
Author	The author of the bug report
Severity	The estimated impact of a bug as perceived by the reporter of the bug. There are several severity labels including `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. Aside from these severity levels, there is one additional severity level that denotes feature requests, i.e., `enhancement`. In this study, we ignore bug reports with this severity label as we focus on defects and not feature requests.
Priority	The priority of a bug to be fixed which is assigned by a bug triager. When the bug report is submitted, this field would be blank. The triager would then decide an appropriate priority level for a bug report. There are five priority levels: P1, P2, P3, P4, and P5.

fields of the bug report. Some bugs are also reported as duplicate bug reports at this point. This is possible due to the distributed nature of the bug reporting process – i.e., users from various parts of the world could encounter the same defect and create different bug reports. We show some example bug reports from Eclipse in Table 2. Note that bug reports shown in the same box (e.g., 4629 and 4664) are duplicates of one another. Eventually a bug triager would forward the bug to a developer to fix it. The developer then works on the bug and eventually comes up with a resolution. The developer may also change the values of some fields of the bug report when working on it.

2.2 Text Pre-Processing

Here we present several standard pre-processing techniques to convert a textual document into a set of features. These pre-processing techniques include tokenization, stop-word removal, and stemming. We present each of them in the following paragraphs.

Tokenization A textual document contains many words. Each of such words is referred to as a token. These words are separated by delimiters which could be spaces, punctuation marks, etc. Tokenization is a process to extract these tokens from a textual document by splitting the document into tokens according to the delimiters.

Stop-Word Removal Not all words are equally important. There are many words that are frequently used in many documents but carry little meaning or useful information. These words are referred to as stop words. There are many of such stop words including "am", "are", "is", "I", "he", etc. These stop words need to be removed from the set of tokens extracted in the previous steps as they might reduce the effectiveness of machine learning or information retrieval solutions due to their skewed distributions. We use a collection of 30 stop words and also standard abbreviations including, "I'm", "that's", etc.

Stemming Words can appear in various forms; in English, various grammatical rules dictate if a root word appears in its singular, plural, present tense, past tense, future tense, or many other forms. Words originating from the same root word but are not identical with

Table 2 Examples of bug reports from eclipse

	ID	Summary	Product	Comp.	Sev.	Prio.
1	4629	Horizontal scroll bar appears too soon in editor (1GC32LW)	Platform	SWT	normal	P4
	4664	StyledText does not compute correct text width (1GELJXD)	Platform	SWT	normal	P2
2	4576	Thread suspend/resume errors in classes with the "same" name	JDT	Debug	normal	P1
	5083	Breakpoint not hit	JDT	Debug	normal	P1
3	4851	Print ignores print	Platform to file option (1GKXC30)	SWT	normal	P3
	5126	StyledText printing should	Platform implement "print to file"	SWT	normal	P3

Comp.=Component. Sev.=Severity. Prio.=Priority one another are semantically related. For example, there is not much difference in meaning between "write" and "writes". In the text mining and information retrieval community, stemming has been proposed to address this issue. Stemming would try to reduce a word to its *ground* form. For example, "working", "worked", and "work" would all be reduced to "work". There are various algorithms that have been proposed to perform stemming. In this work, we use the Porter's stemming algorithm (PorterStemmer 2011) to process the text as it is commonly used by many prior studies, e.g., Menzies and Marcus (2008), Lamkanfi et al. (2010, 2011), and Wang et al. (2008).

2.3 Measuring Similarity of Bug Reports

Various techniques have been proposed to measure the similarity of bug reports. A number of techniques model a bug report as a vector of weighted tokens. Similarity of two bug reports can then be evaluated by computing the Cosine similarity of their corresponding two vectors. These include the work by Jalbert and Weimer (2008), Runeson et al. (2007), and Wang et al. (2008), etc.

One of the most recent approaches proposed to measure the similarity of bug reports is REP by Sun et al. (2011). Their approach extends BM25F (Robertson et al. 2004) which is a state-of-the-art measure for structured document retrieval. In their proposed approach past bug reports that have been labeled as duplicate are used as training data to measure the similarity of two bug reports. Various fields of bug reports are used for comparison including the textual and non-textual contents of bug reports. We use an adapted version of REP to measure the similarity of bug reports. REP includes the comparison of the priority fields of two bug reports to measure their similarity. In our setting, we would like to predict the values of the priority field. Thus, we remove the priority field from REP's analysis as they are unknown for bug reports whose priority levels are to be predicted. We call the resultant REP, REP^-. REP^- only compares the textual (summary and description), product, and component fields of two bug reports to measure their similarity.

2.4 Problem Definition

In this paper, we assume the existence of a bug tracking system with a set of historical bug reports whose priority labels are known. Given these bug reports, we want to learn a model that can predict the priority labels of other bug reports. Thus, when a new bug report is received, the model can be used to recommend the priority label of this new bug report. In short, our problem definition is as follows:

> *Given a new bug report and a bug tracking system, predict the priority label of the new report as either P1, P2, P3, P4, or P5.*

3 Proposed Approach

In this section, we describe our proposed framework. First we present the overall structure of our framework. Next, we zoom into two sub-components of the framework namely feature extraction and classification modules. In the feature extraction module, we extract various features that capture various factors that potentially affect the priority level of a bug report. In the classification module, we propose a new classification engine leveraging linear regression and thresholding to handle imbalanced data.

3.1 Overall Framework

Our framework, named DRONE, is illustrated in Fig. 1. It runs in two phases: training and prediction. There are two main modules: feature extraction module and classification module.

In the training phase, our framework takes as input a set of bug reports with known priority labels. The feature extraction module extracts various features that capture `temporal`, `textual`, `author`, `related-report`, `severity`, and `product` factors that potentially have an effect on the priority level of a bug report. These features are then fed to the classification module. The classification module would produce a discriminative model that could classify a bug report with unknown priority level.

In the prediction phase, our framework takes a set of bug reports whose priority levels are to be predicted. Features are first extracted from these bug reports. The model learned in the training phase is then used to predict the priority levels of the bug reports by analyzing these features.

Our framework has two placeholders: feature extraction and classification module. Various techniques could be put into these placeholders. We describe our proposed feature extraction and classification modules in the following two subsections.

3.2 Feature Extraction Module

The goal of the feature extraction module is to characterize a bug report in several dimensions: `temporal`, `textual`, `author`, `related-report`, `severity`, and

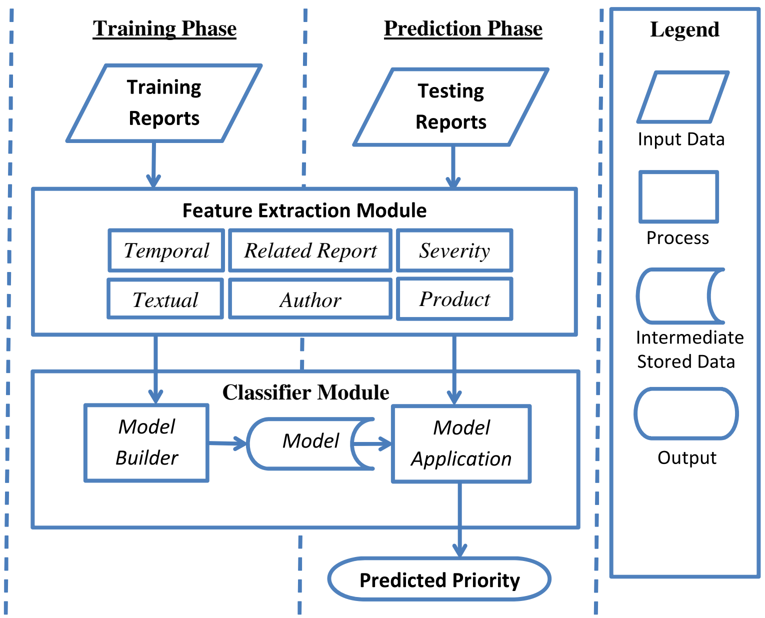


Fig. 1 DRONE framework

product. For each dimension, a set of features is considered. For each bug report *BR* our feature extraction module processes various fields of *BR* and a bug database of reports created prior to the reporting of *BR*. It would then produce a vector of values for the features listed in Table 3.

Each dimension/factor is characterized by a set of features. For the temporal factor, we propose several features that capture the number of bugs that are reported in the last x days with priority level y. We vary the values of x and y to get a number of features (TMP1-12). Intuitively, if there are many bugs reported in the last x days with a higher severity level than *BR*, *BR* is likely not assigned a high priority level since there are many higher severity bug reports in the bug tracking system that need to be resolved too.

For the textual factor, we take the description of the input bug report *BR* and perform the text pre-processing steps mentioned in Section 2. Each of the resultant word tokens corresponds to a feature. For each feature, we take the number of times it occurs in a description as its value. Collectively these features (TXT1-n) describe what the bug is all about and this determines how important it is for a particular bug to get fixed. Term frequency inverse document frequency (TF-IDF) weighting scheme is often used to assign values to a word token (Manning et al. 2008). In this work, we only use term frequency, i.e., the number of times a word token occurs in a description. We have tried to use TF-IDF weighting scheme to assign values, however, we find that the result is worse than using TF only. One possible reason is that IDF is oblivious to the priority labels which can result in "inappropriate scaling for some features" (Forman 2008). For example, a word *w1* appears in 90 % of P1 bug reports whereas *w2* appears in 10 % of P1 bug reports, and both of them never appear in bug reports of other priority labels. IDF gives a large boost to *w2* as it appears much less frequently. However, in this case, *w1* should be a stronger predictor since it appears in 90 % of P1 bug reports.

For the author factor, we capture the mean and median priority, and number of all bug reports that are made by the author of *BR* prior to the reporting of *BR* (AUT1-3). We extract author factor features based on the hypothesis that if an author always reports high priority bugs, he or she might continue reporting high priority bugs. Also, the more bugs an author reports, it is likely that the more reliable his/her severity estimation of the bug would be.

For the related-report factor, we capture the mean and median priority of the top-k reports as measured using REP^{-}. REP^{-} is a bug report similarity measure adapted from the work by Sun et al. (2011) – described in Section 2. We vary the value k to create a number of features (REP1-10). Considering that similar bug reports might be assigned the same priority, we analyze the top-k most similar reports to a bug report *BR* to help us decide the priority of *BR*. For the severity factor, we use the severity field of *BR* as a feature.

For the product factor, we capture features related to the product and component fields of *BR*. The product field specifies a part of the software system that is affected by the issue reported in *BR*. The component field specifies more specific sub-parts of the software system that are affected by the issue reported in *BR*. For each of the product and component fields, we extract 11 features that capture the value of the field (PRO1,PRO12), some statistics of bug reports made for that particular product/component prior to the reporting of *BR* (PRO2-9,PRO13-20), and the mean and median priority levels of bug reports made for that particular product/component prior to the reporting of *BR* (PRO10-11,PRO21-22). Some products or components might play a more major role in the software systems than other

Table 3 DRONE features extracted for a bug report BR

`Temporal` Factor	
TMP1	Number of bugs reported within 7 days before the reporting of *BR*
TMP2	Number of bugs reported with the same severity within 7 days before the reporting of *BR*
TMP3	Number of bugs reported with the same or higher severity within 7 days before the reporting of *BR*
TMP4-6	The same as TMP1-3 except the time duration is 30 days
TMP7-9	The same as TMP1-3 except the time duration is 1 day
TMP10-12	The same as TMP1-3 except the time duration is 3 days
`Textual` Factor	
TXT1-n	Stemmed words from the description field of *BR* excluding stop words (Specifically, n=395,996 in our experiment).
`Author` Factor	
AUT1	Mean priority of all bug reports made by the author of *BR* prior to the reporting of *BR*
AUT2	Median priority of all bug reports made by the author of *BR* prior to the reporting of *BR*
AUT3	The number of bug reports made by the author of *BR* prior to the reporting of *BR*
`Related-Report` Factor	
REP1	Mean priority of the top-20 most similar bug reports to *BR* as measured using REP^- prior to the reporting of *BR*
REP2	Median priority of the top-20 most similar bug reports to *BR* as measured using REP^- prior to the reporting of *BR*
REP3-4	The same as REP1-2 except only the top 10 bug reports are considered
REP5-6	The same as REP1-2 except only the top 5 bug reports are considered
REP7-8	The same as REP1-2 except only the top 3 bug reports are considered
REP9-10	The same as REP1-2 except only the top 1 bug report is considered
`Severity` Factor	
SEV	*BR*'s severity field.
`Product` Factor	
PRO1	*BR*'s product field. This categorical feature is translated into multiple binary features.
PRO2	Number of bug reports made for the same product as that of *BR* prior to the reporting of *BR*
PRO3	Number of bug reports made for the same product of the same severity as that of *BR* prior to the reporting of *BR*
PRO4	Number of bug reports made for the same product of the same or higher severity as those of *BR* prior to the reporting of *BR*
PRO5	Proportion of bug reports made for the same product as that of *BR* prior to the reporting of *BR* that are assigned priority P1.
PRO6-9	The same as PRO5 except they are for priority P2-P5 respectively.
PRO10	Mean priority of bug reports made for the same product as that of *BR* prior to the reporting of *BR*
PRO11	Median priority of bug reports made for the same product as that of *BR* prior to the reporting of *BR*
PRO12-22	The same as PRO1-11 except they are for the component field of *BR*.

products or components – for these products a triager might assign higher priority levels. We extract these 22 `product` features to characterize *BR*'s product and component for a better prediction of its priority level.

We include both mean and median (for `author`, `related-report`, and `product` factors) rather than only one of them because at times the mean and median are different. Overall, most reports have a priority of P3. However, for a product or an author, this might not be true. It can be the case that a bug report author mostly submits critical reports that must be given high priority, or a product is so important that the majority of its bug reports are given high priority. Similarly, for a bug report, the set of its top-N most similar reports might not be dominated by bug reports of priority P3. If the top-N similar reports of two bug reports are mostly P3 reports, their medians will be the same (i.e., P3), but their means will be different. In this case, the mean will provide more discrimination than the median.

3.3 Classification Module

Feature vectors produced by the feature extraction module for the training and testing data would be fed to the classification module. The classification module has two parts corresponding to the training and prediction phases. In the training phase, the goal is to build a discriminative model that could predict the priority of a new bug report with unknown priority. This model would be used in the prediction phase to assign priority levels to bug reports.

In this work, we propose a classification engine named GRAY. We illustrate our classification engine in Fig. 2. It has two main parts: linear regression and thresholding. Our approach utilizes linear regression to capture the relationship between the features and the priority levels. As our data is imbalanced (i.e., most of the bug reports are assigned priority level P3), we employ a *thresholding* approach to calibrate a set of thresholds to decide the class labels (i.e., priority levels).

We follow a regression approach rather than a standard classification approach for the following reason. The bug reports are of 5 priority levels (P1-P5). These priority levels are not categorical values rather they are ordinal values since there is a total ordering among these levels. Level P1 is higher than level P2, which is in turn higher than level P3, and so on. To capture this ordering among levels, we use regression rather than a standard classification approach. Standard classification approaches, e.g., standard support vector machine, naive bayes, logistic regression, etc., consider the class labels to be categorical. Also, many approaches and standard tools only support two class labels: +ve and -ve.

Given a training data, a linear regression approach would build a model capturing the relationship between a set of explanatory variables with a dependent variable. If the set of explanatory variables has more than one member, it is referred to as multiple regression, which is the case for our approach. In our problem setting, the features form the set of explanatory variables while the priority level is the dependent variable. A bug report in the prediction phase would be converted to a vector of features values, which is then treated as a set of explanatory variables. The model learned during linear regression could then be applied to output the value for the dependent variable which is a real number.

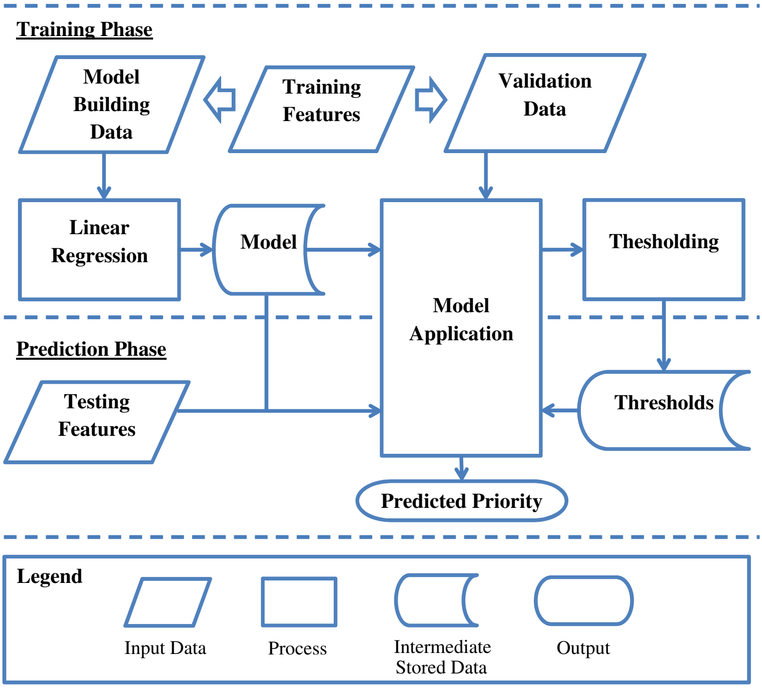


Fig. 2 GRAY classification engine

The next step is to convert the value of the dependent variable to one of the five priority levels. One possibility is to simply truncate the value of the dependent variable to the nearest integer and treat this as the priority level. However, this would not work well for our data as it is imbalanced with most bug reports having priority 3 – thus many of the values of the dependent variable are likely to be close to 3. To address this issue we employ a *thresholding* approach to pick four thresholds to be the boundaries of the five priority levels.

We sort the bug reports in our training data based on their bug identifiers. The bug identifiers are created automatically by Bugzilla in ascending order, based on the time bug reports are received. We take the first 50 % of the bug reports in our dataset as model building data, and use the remainder of the bug reports as the validation data. We do this to ensure that bug reports that are in the model building data are reported earlier than bug reports in the validation data. We do not use cross validation (Han et al. 2011), since we should not use future bug reports to predict past bug reports.

The model building training data is used to train a regression model. The linear regression model is then applied on the validation data which generates a linear regression score for each report. The validation training data is used to infer the four thresholds using our thresholding approach. The pseudocode of this process which employs greedy hill climbing to tune the thresholds is shown in Algorithm 1. The resultant linear regression model and thresholds are then used to classify bug reports in the testing data whose priority level is to be predicted based on their feature vectors.

Algorithm 1 Tune Thresholds Using Greedy Hill Climbing

1: **Input:**
2: $VData$: Validation Data
3: **Output:**
4: T : The four thresholds: T_1, T_2, T_3, and T_4
5: **Method:**
6: Initialize T based on the proportion of reports assigned as P1, P2, P3, P4, and P5 in $VData$ (see text).
7: Let T_0 = minimum regression score of reports in $VData$.
8: **for all** $T_i \in \{T_1, T_2, T_3, T_4\}$ **do**
9:     Let $D = T_i - T_{i-1}$
10:     **repeat**
11:         Try to increase T_i by $1\% \times D$, compute new F-measure on $VData$
12:         Try to decrease T_i by $1\% \times D$, compute new F-measure on $VData$
13:         Update T_i if the increase or decrease improves F-measure and $T_0 < T_1 < T_2 < T_3 < T_4$
14:     **until** T_i is not updated
15: **end for**
16: **return** Tuned thresholds T

We first set the 4 thresholds based on the proportion of bug reports that are assigned as P1, P2, P3, P4, and P5 in the validation data (Line 6). For example, if the proportion of bug reports belonging to P1 in the validation data is only 10 %, then we sort the data points in the validation data based on their linear regression scores, and set the first threshold as the regression output of the data point at the 10th percentile. Next, we modify each thresholds one by one to achieve higher F-measure (Lines 8-15). For each threshold level, we try to increase it or decrease it by a small step, which is 1 % of the distance between a threshold level to the previous threshold level (Lines 9, 11-12). At each step, after we change the threshold level, we evaluate if the resultant threshold levels could increase the average F-measure for the validation data points or not. If it is, we will keep the new threshold level otherwise we will discard the new threshold level (Line 13). We continue the process until we can no longer improve the average F-measure by moving a threshold level, with a constraint that a threshold cannot be moved beyond the next threshold level or under the previous threshold level, i.e., the second threshold cannot be set higher than the third threshold (Line 14).

4 Experiment Setup

In this section, we first describe four scenarios where we apply and evaluate DRONE. We then describe datasets that we use to investigate the effectiveness of DRONE. Next, we present our experimental setting and evaluation measures. Finally, we present our research questions.

4.1 Scenarios

The value of the various fields in a bug report can be changed while the bug report is processed by triagers and developers. Fields can be changed for various reasons. One reason is the initial values of the fields are incorrect. Based on this observation, we consider four different scenarios:

Last In this scenario, we predict the *last* value of the priority field given the *last* values of other fields in the bug report. We evaluate the effectiveness of our approach when the values of all other fields have been finalized.

Assigned In this scenario, we predict the value of the priority field, given the values of other fields, at the time a bug report status is changed to "Assigned". When a bug report is received, its status is typically "Unconfirmed" or "New". After some checks, if it is valid, following standard procedure, its status would eventually be changed to "Assigned" indicating that a bug report has been assigned to a developer and the assigned developer is working on the report. At this point, the values of bug report fields are likely to be more reliable.

First In this scenario, we predict the *first* value of the priority field given the *first* values of other fields in a bug report. This scenario is meant to evaluate how accurate our approach is considering noisy values of initial bug report fields (i.e., they might get changed later).

No-P3 This scenario is similar to scenario "Last". The only difference is that we remove all bug reports whose priority levels are "P3" (i.e., the default priority level). Since P3 is the default value of the priority field, it *might* be the case that for P3 bug reports, developers do not put much thought when setting the priority level. However, most bug reports are assigned P3. Thus, deleting these bug reports would mean omitting the majority of bug reports. Due to the pros and cons of excluding (or including) P3 bug reports, we investigate *both* "Last" and "No-P3" scenarios.

Mixed This scenario mixes scenario "First" and scenario "Last". In this scenario, we want to investigate if we train DRONE with the best possible data, will it be able to correctly predict the priority of a new bug report when the values of its fields are likely to be noisy

4.2 Dataset Collection

We investigate the bug repository of Eclipse. Eclipse is an integrated development platform to support various aspects of software development. It is a large open source project that is supported and used by many developers around the world. In the following paragraphs, we describe how we collect an Eclipse dataset for each of the 4 scenarios described in the previous sub-section.

4.2.1 Scenario "Last"

We consider the bug reports submitted from October 2001 to December 2007 and download them from Bugzilla[1]. We collect only defect reports and ignore those that correspond to feature requests. Since these bug reports were submitted many years back (6-12 years back), the values of various fields in the reports are unlikely to be changed further. These reports contain the last values of the fields after modifications (if any).

[1] https://bugs.eclipse.org/bugs/

Table 4 Eclipse dataset details

Period		REP$^-$ Train.		DRONE Train.	Test.
From	To	#Duplicate	#All	#All	#All
2001-10-10	2007-12-14	200	3,312	87,649	87,648

Train.=Training Reports Test.=Testing Reports

We sort the bug reports in chronological order. We divide the dataset into three: REP$^-$ training data, DRONE training data, and the test data. The REP$^-$ training data is the first N reports containing 200 duplicate bug reports (c.f. Sun et al. 2011). This data is used to train the parameters of REP$^-$ such that it is better able to distinguish similar bug reports. We split the remaining data into DRONE training and testing data. We use the first half of the bug reports (sorted in chronological order) for training and keep the other half for testing. We separate training data and testing data based on chronological order to simulate the real setting where our approach would be used. This evaluation method is also used in many other research studies that also analyze bug reports (Hiew 2006; Nguyen et al. 2012; Runeson et al. 2007). We show the distribution of bug reports used for training and testing in Table 4. We plot the distribution of the priorities of these bug reports in Fig. 3.

4.2.2 *Scenario "Assigned" and "First"*

The datasets used for scenario "Assigned" and "First" are similar to the dataset used for scenario "Last". However, rather than using the last values of the various fields in the bug reports, we need to reverse engineer the values of the fields when the bug report status was changed to assigned (for Scenario "Assigned") and the values of the fields when the bug report was submitted (for Scenario "First"). In order to obtain the values of the priority and other fields of bug reports for scenario "First" and "Assigned", we investigate the modification histories of bug reports. A modification history of a bug report specifies for each modification: the person who made the modification, the time when the modification was performed, the fields whose values get modified, the values that get deleted, and the values that get added.

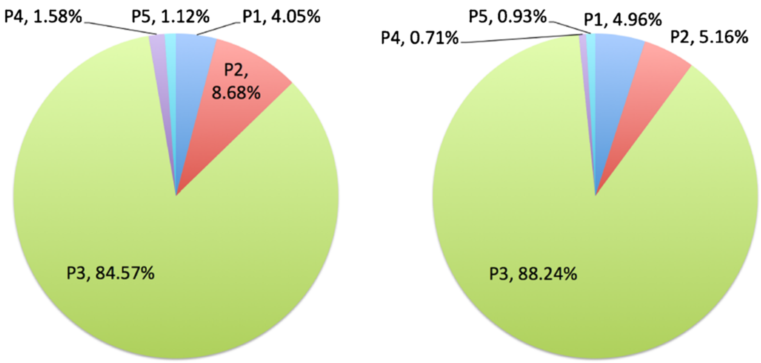


Fig. 3 Distribution of priorities of training reports (*left*) and testing reports (*right*)

Table 5 Modification history for bug report with Id 5110

Who	When	What	Removed	Added
James_Moody	2001-10-26 11:28:58 EDT	Assignee	Kevin_McGuire	James_Moody
James_Moody	2001-10-26 14:21:46 EDT	CC		Kevin_McGuire
James_Moody	2001-11-01 12:07:54 EST	Status	NEW	ASSIGNED
James_Moody	2002-01-03 16:42:56 EST	Priority	P3	P5
Kevin_McGuire	2002-04-17 17:18:09 EDT	Status	ASSIGNED	RESOLVED
		Resolution	—	FIXED
Kevin_McGuire	2002-05-23 21:20:40 EDT	Target Milestone	—	2.0 M6

An example of a modification history of a bug report is shown in Table 5. The modification history specifies that five modifications have been performed. The first modification was performed by James_Moody at 11.28 am EDT on the 26th of October 2001. James_Moody changed the value of the assignee field to himself. The third modification, on the 1st of November 2001, changed the status from new to assigned indicating that he starts working on the bug report. Around two months later, on the 3rd of January 2002, the priority is changed from P3 to P5. This illustrates a bug report where the priority level considered by Scenario "Last" (i.e., P5) differs from the priority level considered by Scenario "Assigned" (i.e., P3).

Another example of a modification history of a bug report is shown in Table 6. The modification history specifies that three modifications have been performed. The first modification was performed by paules at 10.35 pm EDT on the 2nd of May 2007. The developer paules changed two fields: status, summary and target milestone. The status was changed from new to assigned indicating that paules started working on the problem. At the same time, paules changed the value of the priority field from P3 to P1. Also, paules added a target milestone which is 4.4i3. This illustrates a bug report where the priority level considered by Scenario "Last" and "Assigned" (i.e., P1) differs from the priority level considered by Scenario "First" (i.e., P3).

4.2.3 Scenario "No-P3"

The dataset used for scenario "No-P3" is similar to the dataset used for scenario "Last". However, we remove bug reports whose final priority levels are P3 from original DRONE training and testing bug reports. The resultant dataset contains 23,830 bug reports, where 13,529 bug reports are used as training reports and 10,301 bug reports are used as testing reports.

Table 6 Modification history for bug report with Id 185222

Who	When	What	Removed	Added
paules	2007-05-02 22:35:49 EDT	Status	NEW	ASSIGNED
		Priority	P3	P1
		Target Milestone	—	4.4i3
paules	2007-05-03 08:08:19 EDT	Status	ASSIGNED	RESOLVED
		Resolution	—	FIXED
jptoomey	2007-07-11 12:37:50 EDT	Status	RESOLVED	CLOSED

4.2.4 Scenario "Mixed"

We train DRONE using the last values of the various fields in the bug reports following scenario "Last". However, we apply DRONE to predict the priority of new bug reports based on the values of the various fields when the report was first submitted.

4.3 Baseline Approaches

We compare our approach with an adapted version of Severis which was proposed by Menzies and Marcus (2008). Severis predicts the severity of bug reports. In the adapted Severis, we simply use it to predict the priority of bug reports. We use the same feature sets and the same classification algorithm described in the Menzies and Marcus's paper. Following the experimental setting described in their paper, we use the top 100 word token features (in terms of their information gain) as it has been shown to perform best among the other options presented in their paper. We refer to the updated Severis as Severis^{Prio}. We also add severity label as an additional feature to Severis^{Prio} and refer to the resultant solution Severis^{Prio+}. We compare Severis^{Prio} and Severis^{Prio+} to our proposed framework DRONE. All experiments are run on an Intel Xeon X5675 3.07GHz server, having 128.0GB RAM, and running Windows Server 2008 operating system.

4.4 Evaluation Measures

Precision, recall, and F-measure, which are commonly used to measure the accuracy of classification algorithms, are used to evaluate the effectiveness of DRONE and our baseline approaches: Severis^{Prio} and Severis^{Prio+}. We evaluate the precision, recall, and F-measure for each of the priority levels. This follows the experimental setting of Menzies and Marcus to evaluate Severis (Menzies and Marcus 2008). The definitions of precision, recall, and F-measure for a priority level P are given below:

$$prec(P) = \frac{Number\ of\ priority\ P\ reports\ correctly\ labeled}{Number\ of\ reports\ labeled\ as\ priority\ level\ P}$$

$$recall(P) = \frac{Number\ of\ priority\ P\ reports\ correctly\ labeled}{Number\ of\ priority\ P\ reports}$$

$$F - measure(P) = 2 \times \frac{precision \times recall}{precision + recall}$$

4.5 Research Questions

For the first scenario (Scenario "Last"), we consider three research questions:

- RQ1 How accurate is our proposed approach as compared with the baseline approaches namely SeverisPrio and SeverisPrio+?
- RQ2 Which of the features are the most effective in discriminating the priority levels?
- RQ3 What are the effectiveness of various classification algorithms in comparison with GRAY in predicting the priority levels of bug reports?

For the other scenarios, since answers to RQ2 and RQ3 are likely to be similar to answers for the first scenario, we only focus on answering RQ1.

Table 7 Precision, recall, and f-measure for DRONE (Scenario "Last")

Priority	Precision	Recall	F-Measure
P1	29.54 %	32.95 %	31.15 %
P2	12.56 %	17.97 %	14.78 %
P3	91.11 %	86.25 %	88.62 %
P4	4.19 %	8.04 %	5.51 %
P5	4.08 %	9.62 %	5.72 %
Average	28.30 %	30.97 %	29.16 %

5 Experiment Results

Here, we present the answers to the three research questions. The first one compare DRONE with Severis^{Prio} and $Severis^{Prio+}$ on accuracy. Next, we zoom in to the various factors that influence the effectiveness of DRONE. In particular, we inspect the features that are most discriminative. We also replace the classification module of DRONE with several other classifiers and investigate their effects on the accuracy of the resultant approach.

5.1 RQ1: Accuracy of DRONE vs. Accuracy of Baselines

5.1.1 Results for Scenario "Last"

The result of DRONE is shown in Table 7. We note that the F-measures are better for P1, P2, and P3 priority levels than for P4, and P5 priority levels.

The result for Severis^{Prio} is shown in Table 8. We note that the F-measures of Severis^{Prio} are zeros for P1, P2, P4, and P5 as it does not assign any bug report correctly in any of these priority levels. Comparing these with the result of DRONE (in Table 7), we note that we can improve the average of the F-measures by a relative improvement of 55.52 % (i.e., $(29.16 - 18.75)/18.75 \times 100$ %). Thus, clearly DRONE performs better than Severis^{Prio}. We believe in report prioritization higher accuracy for high priority bugs (i.e., P1 and P2) is much more important than higher accuracy for low priority bugs (i.e., P3, P4, and P5) because identifying them can help developers fix the most important bug reports first. We also believe that higher accuracy for bug reports with priority P4 and P5 is more important than higher accuracy for bug reports with priority P3. This is the case since developers can safely set bug reports with priority P4 and P5 aside (they are unimportant) and fix them when time permits, which can improve the overall efficiency. On the other hand, since the majority of bug reports are P3 reports, developers can neither prioritize or safely set them aside.

Table 8 Precision, recall, and f-measure for Severis^{Prio} (Scenario "Last")

Priority	Precision	Recall	F-Measure
P1	0.00 %	0.00 %	0.00 %
P2	0.00 %	0.00 %	0.00 %
P3	88.25 %	100.00 %	93.76 %
P4	0.00 %	0.00 %	0.00 %
P5	0.00 %	0.00 %	0.00 %
Average	17.65 %	20.00 %	18.75 %

Table 9 Precision, recall, and f-measure for Severis^{Prio+} (Scenario “Last”)

Priority	Precision	Recall	F-Measure
P1	0.00 %	0.00 %	0.00 %
P2	0.00 %	0.00 %	0.00 %
P3	88.25 %	100.00 %	93.76 %
P4	0.00 %	0.00 %	0.00 %
P5	0.00 %	0.00 %	0.00 %
Average	17.65 %	20.00 %	18.75 %

Table 10 Precision, recall, and f-measure for DRONE (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	34.68 %	39.31 %	36.86 %
P2	12.52 %	19.70 %	15.31 %
P3	90.75 %	82.71 %	86.54 %
P4	3.75 %	12.96 %	5.82 %
P5	2.73 %	7.27 %	3.97 %
Average	28.89 %	32.39 %	29.70 %

Table 11 Precision, recall, and f-measure for Severis^{Prio} (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	0 %	0 %	0 %
P2	0 %	0 %	0 %
P3	86.27 %	99.86 %	92.57 %
P4	0 %	0 %	0 %
P5	0 %	0 %	0 %
Average	17.25 %	19.97 %	18.51 %

Table 12 Precision, recall, and f-measure for Severis^{Prio+} (Scenario “Assigned”)

Priority	Precision	Recall	F-Measure
P1	17.07 %	0.63 %	1.22 %
P2	23.33	0.56 %	1.09 %
P3	86.31 %	99.68 %	92.51 %
P4	0 %	0 %	0 %
P5	0 %	0 %	0 %
Average	25.34 %	20.17 %	18.96 %

The result for Severis^{Prio+} is shown in Table 9. We note that the result of Severis^{Prio+} is the same as Severis^{Prio}. Thus, our proposed approach DRONE also outperforms Severis^{Prio+}.

5.1.2 Results for Scenario "Assigned"

Here, we present the answer to the first research question for scenario "Assigned". The result of DRONE is shown in Table 10. We note that the F-measures are better for P1, P2, and P3 priority levels than for P4, and P5 priority levels.

The result for Severis^{Prio} is shown in Table 11. We note that the F-measures of Severis^{Prio} are zeros for P1, P2, P4, and P5 as Severis^{Prio} predicts most of these bug reports as P3 priority level. Comparing these with the result of DRONE (in Table 10), we note that we can improve the average of the F-measures by a relative improvement of 60.45 %. Thus, DRONE performs better than Severis^{Prio}.

The result for Severis^{Prio+} is shown in Table 12. We note that the result of Severis^{Prio+} is a little better than Severis^{Prio}. But the performance of Severis^{Prio+} is still worse than our proposed approach DRONE. DRONE can improve the average F-measure by a relative improvement of 56.64 %.

5.1.3 Results for Scenario "First"

Here, we present the answer to the first research question for scenario "First". The result of DRONE is shown in Table 13. We note that the F-measures of DRONE for P1, P2, P4 and P5 are all zeros because it has predicted almost every bug report as P3 priority level.

The results for Severis^{Prio} and Severis^{Prio+} are shown in Tables 14 and 15. We note that these two approaches have similar results as DRONE. One reason why the performance of all approaches are worse for scenario "First" is that *almost all* of the bug reports are initialized with priority P3, which is the default value.

5.1.4 Results for Scenario "No-P3"

Here, we present the answer to the first research questions for scenario "No-P3". The result of DRONE is shown in Table 16. We note that the F-measures are better for P1, P2 than for P4 and P5.

The result for Severis^{Prio} is shown in Table 17. Comparing these with the result of DRONE (in Table 16), we note that we can improve the average F-measure by a relative improvement of 209 %. Thus, clearly DRONE performs better than Severis^{Prio}.

The result for Severis^{Prio+} is shown in Table 18. We note that our approach DRONE still performs better than Severis^{Prio+}, with a relative improvement of 35 %.

Table 13 Precision, recall, and f-measure for DRONE (Scenario "First")

Priority	Precision	Recall	F-Measure
P1	0 %	0 %	0 %
P2	0 %	0 %	0 %
P3	99.99 %	99.85 %	99.92 %
P4	0 %	0 %	0 %
P5	0 %	0 %	0 %
Average	20.00 %	19.97 %	19.98 %

Table 14 Precision, recall, and f-measure for Severis^{Prio} (Scenario "First")

Priority	Precision	Recall	F-Measure
P1	0 %	0 %	0 %
P2	0 %	0 %	0 %
P3	100 %	100 %	100 %
P4	0 %	0 %	0 %
P5	0 %	0 %	0 %
Average	20 %	20 %	20 %

Table 15 Precision, recall, and f-measure for Severis^{Prio+} (Scenario "First")

Priority	Precision	Recall	F-Measure
P1	0 %	0 %	0 %
P2	0 %	0 %	0 %
P3	100 %	100 %	100 %
P4	0 %	0 %	0 %
P5	0 %	0 %	0 %
Average	20 %	20 %	20 %

Table 16 Precision, recall, and f-measure for DRONE (Scenario "No-P3")

Priority	Precision	Recall	F-Measure
P1	69.78 %	64.49 %	67.03 %
P2	61.04 %	63.56 %	62.27 %
P4	10.88 %	7.56 %	8.92 %
P5	46.98 %	66.21 %	54.96 %
Average	47.17 %	50.46 %	48.30 %

Table 17 Precision, recall, and f-measure for Severis^{Prio} (Scenario "No-P3")

Priority	Precision	Recall	F-Measure
P1	54.17 %	0.60 %	1.18 %
P2	43.93 %	99.42 %	60.94 %
P4	12.50 %	0.16 %	0.32 %
P5	0 %	0 %	0 %
Average	27.65 %	25.04 %	15.61 %

Table 18 Precision, recall, and f-measure for Severis^{Prio+} (Scenario "No-P3")

Priority	Precision	Recall	F-Measure
P1	74.33 %	26.20 %	38.75 %
P2	48.76 %	90.39 %	63.35 %
P4	51.87 %	31.19 %	38.96 %
P5	77.78 %	0.86 %	1.7 %
Average	63.18 %	37.16 %	35.69 %

5.1.5 Results for Scenario "Mixed"

The result of DRONE is shown in Table 19. We note that the F-measures are better for P1, P2, and P3 priority levels than for P4, and P5 priority levels.

The result for Severis^{Prio} is shown in Table 20. We note that the F-measures of Severis^{Prio} are zeros for P1, P2, P4, and P5 as it does not assign any bug report correctly in any of these priority levels. Comparing these with the result of DRONE (in Table 19), we note that we can improve the average of the F-measures by a relative improvement of 38.35 %. Thus, clearly DRONE performs better than Severis^{Prio}.

The result for Severis^{Prio+} is shown in Table 21. We note that the result of Severis^{Prio+} is the same as Severis^{Prio}. Thus, our proposed approach DRONE also outperforms Severis^{Prio+}.

5.2 RQ2: Most Discriminative Features

Next, we would like to find the most discriminative features among the 20,000+ features that we have (including the word tokens). Information gain (Manning et al. 2008) and Fisher score (Duda et al. 2000) are often used as discriminativeness measures. Since many of the features are non-binary features, we use Fisher score as it captures the differences in the distribution of the feature values across the classes (i.e., the priority levels).

At times features that are only exhibited in a few data instances receive high Fisher score. This is true for the word tokens. However, these are not good features as they appear too sparsely in the data. Thus we focus on features that appear in at least 0.5 % of the data. For these features, Table 22 shows the top-10 features sorted according to their Fisher score (the higher the better). We notice that six of them are features related to `product` factor and three of them are features related to `related-report` factor. It suggests that the product a bug report is about and existing related reports influence the priority label assigned to the report.

Table 19 Precision, recall, and f-measure for DRONE (Scenario "Mixed")

Priority	Precision	Recall	F-Measure
P1	23.07 %	23.23 %	23.15 %
P2	10.16 %	12.84 %	11.34 %
P3	89.98 %	86.91 %	88.42 %
P4	1.53 %	2.57 %	1.92 %
P5	3.54 %	7.89 %	4.89 %
Average	25.66 %	26.69 %	25.94 %

Table 20 Precision, recall, and f-measure for Severis^{Prio} (Scenario "Mixed")

Priority	Precision	Recall	F-Measure
P1	0.00 %	0.00 %	0.00 %
P2	0.00 %	0.00 %	0.00 %
P3	88.25 %	100.00 %	93.76 %
P4	0.00 %	0.00 %	0.00 %
P5	0.00 %	0.00 %	0.00 %
Average	17.65 %	20.00 %	18.75 %

Table 21 Precision, recall, and f-measure for Severis^{Prio+} (Scenario "Mixed")

Priority	Precision	Recall	F-Measure
P1	0.00 %	0.00 %	0.00 %
P2	0.00 %	0.00 %	0.00 %
P3	88.25 %	100.00 %	93.76 %
P4	0.00 %	0.00 %	0.00 %
P5	0.00 %	0.00 %	0.00 %
Average	17.65 %	20.00 %	18.75 %

Table 22 Top-10 features in terms of fisher score (Scenario "Last")

Rank	Feature name	Fisher score
1	PRO5	0.142
2	PRO16	0.132
3	REP1	0.109
4	REP3	0.101
5	PRO18	0.092
6	PRO10	0.091
7	PRO21	0.088
8	PRO7	0.088
9	REP5	0.087
10	"1663"	0.079

Table 23 Comparisons of average f-measures of GRAY versus other classifiers (Scenario "Last")

Class.	F-Measures					
	P1	P2	P3	P4	P5	Ave.
GRAY	31.15 %	14.78 %	88.62 %	5.51 %	5.72 %	29.16 %
SM	0 %	0 %	93.76 %	0 %	0 %	18.75 %
RIPPER	CC	CC	CC	CC	CC	CC
NBM	OOM	OOM	OOM	OOM	OOM	OOM

Class. = Classifiers SM = SVM-MultiClass NBM = Naive Bayes Multinomial OOM = Out-Of-Memory (more than 9GB) CC = cannot complete in time (more than 8 hours)

We notice that the 10^{th} most discriminative feature is a word token “1663”. This token comes from a line in various stack traces included in many bug reports which is:

org.eclipse.ui.internal.Workbench.run(Workbench.java:1663)

It is discriminative as it appears in 15 % of the bug reports assigned priority level P5, while it only appears in 0.77, 1.29, 0.99, and 0.00 % of the bug reports assigned priority level P1, P2, P3, and P4 respectively. It seems the inclusion of stack traces that include the above line enables developers to identify P5 bugs better.

5.3 RQ3: Effectiveness of Various Classification Algorithms

The classification engine of our DRONE framework could be replaced with other classification algorithms aside from GRAY. We experiment with several classification algorithms (SVM-MultiClass Crammer and Singer (2001), RIPPER Cohen (1995), and Naive Bayes Multinomial Manning et al. (2008)) and compare their F-measures across the five priority levels with GRAY. We use the implementation of SVM-MultiClass available from SVM-MultiClass (2011). We use the implementations of RIPPER and Naive Bayes Multinomial in WEKA (2011). We show the result in Table 23. We notice that in terms of average F-Measures GRAY outperforms SVM-MultiClass by a relative improvement of 55.52 %. Naive Bayes Multinomial is unable to complete due to an out-of-memory exception although we have allocated more than 9GB of RAM to the JVM in our server. RIPPER could not complete after running for more than 8 hours.

We also perform an experiment where we only use 10 % of the training data (the last 10 % of the training data (i.e., the most recent training data)) for GRAY, SM, RIPPER, and NBM and evaluate the resultant learned models on the test data. The results are shown in Table 24. We find that GRAY outperforms the other 3 classifiers. The best competitor is RIPPER; it requires the longest amount of time to build its model (i.e., about 1.5 hours).

6 Discussion

In this section, we first present the threats to validity. Next, we explain the reasons that might cause the bad performance of the baseline approaches. Finally, we compare our model with a trivial approach that simply assigns a bug report to the most common priority level.

Table 24 Comparisons of average f-measures of GRAY versus other classifiers using 10 % of the training data (Scenario “Last”)

Class.	F-Measures					
	P1	P2	P3	P4	P5	Ave.
GRAY	36.17 %	14.25 %	81.45 %	0.80 %	5.41 %	27.62 %
SM	0 %	0 %	93.76 %	0 %	0 %	18.75 %
RIPPER	32.40 %	3.50 %	93.50 %	1.00 %	1.40 %	26.36 %
NBM	0.20 %	1.80 %	93.40 %	0.00 %	0.00 %	19.08 %

Class. = Classifiers SM = SVM-MultiClass NBM = Naive Bayes Multinomial

6.1 Threats to Validity

Threats to internal validity relates to experimental errors. We have checked our implementation and results. Still, there could be some errors that we did not notice. Also, in our experiments, we use the original values of the bug reports fields as assigned by developers. The values of these fields might be wrongly or subjectively assigned by developers. For example, severity is a subjective field in a bug report. Threats to external validity refers to the generalizability of our findings. We consider the repository of Eclipse containing more than a hundred thousand bugs which are reported in a period of more than 6 years. Still, we have only analyzed bug reports from one software system. We exclude some other Bugzilla datasets from two other software systems that we have as most of the reports there do not contain information on the priority field. In the future, we plan to extend our study by considering more programs and bug reports. Threats to construct validity relates to the suitability of our evaluation measures. We use precision, recall, and F-measure which are standard metrics used for evaluating classification algorithms. Also, these measures are used by Menzies and Marcus to evaluate Severis (Menzies and Marcus 2008).

6.2 Dismal Performance of Baseline

Results of Severis^{Prio} and Severis^{Prio+} shown in Tables 8 and 9 are poor. This might be due to a few factors:

1. **(Treating Ordinal Data as Categorical Data)** The RIPPER classification algorithm used by Severis considers the class labels as categorical data. RIPPER, as well as other standard classification algorithms (e.g., SVM, etc), does not consider how different a pair of class labels is as compared to other pairs of class labels. In our setting, RIPPER simply treats P1, P2, P3, P4, and P5 as different labels. P1 is as different to P2, as P1 to P5. We know that this is not the case. Bug reports labeled as P1 are likely to be more similar to those labeled as P2, than those labeled as P5.

   Our approach that enhances linear regression treats class labels as ordinal data. Thus, in our setting P1 is closer to P2 than it is to P5.
2. **(Data Imbalance)** Data imbalance might also negatively affect the performance of the baseline approaches. There are much more bug reports assigned as P3 (74,210 out of 87,649) in the training data; this might make the classifier "think" that the best label to assign to *any* bug report is P3. We solve this problem by a *thresholding* approach that varies the ranges of regression output values, corresponding to each class labels, based on a set of validation data points.

6.3 Comparison with a Trivial Approach

Besides comparing our approach and those adapted from a previous work, we also compare our approach with a trivial approach which simply labels every report as a P3 report. The results of this trivial approach for scenarios "Last" and "First" are almost the same as those of Severis^{Prio} and Severis^{Prio+}. For scenario "Assigned", this trivial approach can predict the P1, P2, P3, P4, and P5 priority levels by F-measures of 0, 0, 92.63, 0, and 0 % respectively. The average of the F-measures is 18.53 %, which is lower than that of DRONE (i.e., 29.16 %). Thus our approach can better predict the priority of bug reports. In particular, our approach performs much better in predicting high priority bug reports that are more important than lower priority ones.

We also compare our approach with a stronger trivial approach that randomly assigns priorities to bug reports based on the proportion of bug reports in the training data that are of priority label P1, P2, P3, P4, and P5. For example, if 90 % of bug reports are of priority label P3, the likelihood of this approach to assign P3 to a new bug report is 90 %. We find that its average F-measures for scenarios “Last”, “First”, “Assigned”, and “Mixed” are 19.32, 19.80, 19.84, and 19.32 % respectively. These are lower than the F-measures for DRONE.

6.4 Impact of Inaccuracies of the Learned Thresholds

Our approach learns thresholds based on a validation data. The validation data is different from the test data and thus the learned thresholds might not be optimal. To check the effect of inaccuracies of the learned thresholds on the validation data, we need to infer ideal thresholds (for the test data) and compare the performance of using ideal thresholds and using learned thresholds.

It is hard to infer ideal thresholds as it involves evaluation of $O(N^4)$ threshold combinations, where N is the size of the test data (which is more than 80,000). To compute a proxy to the ideal thresholds, we perform the following steps:

1. We start with the learned thresholds (from the validation data).
2. We fine tune the learned thresholds on the test data (with the ground truth priority labels). To do this, we perform a greedy search by incrementally increasing or decreasing the values of the thresholds and investigate the impact of these thresholds on the accuracy (average F-measure) of the approach on the test data. We continue to tune these thresholds until the accuracy of our approach cannot be improved further. We follow the process described in Section 3.3.

Considering scenario “Last”, we find that the average F-measure achieved by using the tuned thresholds (i.e., tuned on the test data) is 29.43 %. This is only slightly higher than the average F-measure using the learned thresholds (i.e., learned from the validation data) which is 29.16 %. The result shows that inaccuracies of the learned thresholds have minimal impact on the effectiveness of our proposed approach.

7 Related Work

In this section, we first highlight past studies on bug severity prediction and crash priority prediction. We then describe studies on duplicate bug report detection. Finally, we present other studies that also analyze bug reports in various ways. Our survey here is by no means complete.

7.1 Past Studies on Bug Severity and Crash Priority Prediction

Menzies and Marcus are the first to predict the severity of bug reports (Menzies and Marcus 2008). They analyze the severity labels of various bugs reported in NASA. They propose a technique that analyzes the textual contents of bug reports and outputs *fine-grained* severity levels – one of the 5 severity labels used in NASA. Their approach extracts word tokens from the description of the bug reports. These word tokens are then pre-processed by removing stop words and performing stemming. Important word tokens are then selected based on their information gain. Top-k tokens are then used as features to characterize each bug report. The set of feature vectors from the training data is then fed into a classification algorithm named RIPPER (Cohen 1995). RIPPER would learn a set of rules which are then used to classify future bug reports with unknown severity labels.

Lamkanfi et al. extend the work by Menzies and Marcus to predict severity levels of reports in open source bug repositories (Lamkanfi et al. 2010). Their technique predicts if a bug report is severe or not. Bugzilla has six severity labels including `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. They drop bug reports belonging to the category `normal`. The remaining five categories are grouped into two groups – severe and non-severe. Severe group includes `blocker`, `critical` and `major`. Non-severe group includes `minor` and `trivial`. Thus they focus on the prediction of *coarse-grained* severity labels.

Extending their prior work, Lamkanfi et al. also try out various classification algorithms and investigate their effectiveness in predicting the severity of bug reports (Lamkanfi et al. 2011). They tried a number of classifiers including Naive Bayes, Naive Bayes Multinomial, 1-Nearest Neighbor, and SVM. They find that Naive Bayes Multinomial perform the best among the four algorithms on a dataset consisting of 29,204 bug reports.

Recently, Tian et al. also predict the severity of bug reports by utilizing a nearest neighbor approach to predict fine grained bug report labels (Tian et al. 2012). Different from the work by Menzies and Marcus which analyzes a collection of bug reports in NASA, Tian et al. apply the solution on a larger collection of bug reports consisting of more than 65,000 Bugzilla reports.

Our work is orthogonal to the above studies. Severity labels are reported by users, while priority levels are assigned by developers. Severity labels correspond to the impact of the bug on the software system as perceived by users while priority levels correspond to the importance "a developer places on fixing the bug" in the view of other bug reports that are received (Eclipse 2012).

Khomh et al. automatically assign priorities to Firefox crash reports in Mozilla Socorro server based on the frequency and entropy of the crashes (Khomh et al. 2011). A crash report is automatically submitted to the Socorro server when Firefox fails and it contains a stack trace and information about the environment to help developers debug the crash. In our study, we investigate bug reports that are manually submitted by users. Different from a crash report, a bug report contains natural language descriptions of a bug and might not contain any stack trace or environment information. Thus, different from Khomh et al.'s approach, we employ a text mining based solution to assign priorities to bug reports.

7.2 Duplicate Bug Report Detection

There are a number of approaches proposed to detect duplicate bug reports. Many of these approaches rely on a good similarity measure to find bug reports that are close to one another. These include work by Runeson et al. (2007), Wang et al. (2008), Jalbert and Weimer (2008), Sun et al. (2010, 2011), and many more.

These studies represent a bug report as a vector of feature values extracted from the various fields of the bug report. These vectors of feature values are then compared with one another and a similarity score is computed. All of these studies consider the textual description available in bug reports. Many of them make use of the concepts of term frequency and inverse document frequency to determine the importance of the word tokens appearing in bug reports. The work by Wang et al. (2008) considers execution traces in addition to words in the bug reports; they have shown that execution traces, if present in bug reports, could be used to detect duplicate bug reports accurately. The work by Jalbert and Weimer (2008) and

Sun et al. (2011) consider other non-textual fields in the bug reports, e.g., product, etc., to measure the similarity of two bug reports.

In this study, we make use of prior studies on duplicate bug report detection to measure the similarity of bug reports. In particular we use the measure proposed by Sun et al. (2011).

7.3 Other Studies Analyzing Bug Reports

There are many other families of studies that analyze bug reports. We highlight some of them in the following paragraphs.

A number of studies try to predict the time it takes to fix a bug. Kim and Whitehead perform an empirical study on the time needed to fix bugs (Kim and Whitehead 2006). Relevant statistics like average bug fixing time, the distribution of bug fixing time, and the files with the highest bug fixing time are reported. Weiß et al. propose an automated approach that predicts the number of developer hours needed to fix a bug by analyzing the time needed to fix similar bugs (Weiß et al. 2007). One of the latest studies in this area is by Hosseini et al. (2012).

A number of studies provide bug triaging support by recommending appropriate developers to fix a particular bug. These include the work by Cubranic and Murphy (2004), Anvik and Murphy (2011), Jeong et al. (2009), Tamrawi et al. (2011), and Bhattacharya et al. (2012).

A number of studies group bugs into different categories. Huang et al. propose a text mining solution to categorize bug reports as either capability, security, performance, reliability, requirement, or usability related bugs (Huang et al. 2011). Gegick et al. categorize bug reports as security-related or non-security related (Gegick et al. 2010).

8 Conclusion and Future Work

In this work, we propose a framework named DRONE to predict the priority levels of bug reports in Bugzilla. We consider multiple factors including: `temporal`, `textual`, `author`, `related-report`, `severity` and `product`. These features are then fed to a classification engine named GRAY built by combining linear regression with a thresholding approach to address the issue with imbalanced data and to assign priority labels to bug reports. We have compared our approach with several baselines based on the state-of-the-art study on bug severity prediction by Menzies and Marcus (2008). The result on a dataset consisting of more than 100,000 bug reports from Eclipse shows that our approach outperforms the baselines in terms of average F-measure by a relative improvement of up to 209 % (Scenario "No-P3").

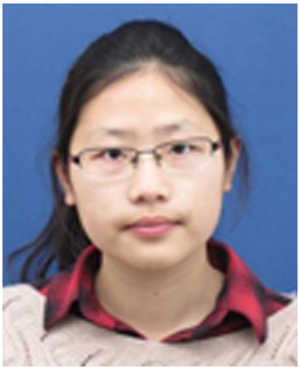
Developers can use our tool as a recommender system to prioritize bugs to be fixed. In the future, we plan to integrate our tool to Bugzilla or other open source bug tracking systems to make it easier for developers to adopt our solution. Also, we plan to include more bug reports from more open source projects to experiment with. We also plan to further improve the accuracy of our approach.

Acknowledgments We would like to thank Serge Demeyer and Foutse Khomh for their comments and advice during our ICSM'13 paper presentation and in the subsequent email exchanges. Their comments and advice motivate us to consider the three additional scenarios: "Assigned", "First", and "No-P3". We would also like to acknowledge Kun Mei, Shaowei Wang, Yang Feng, Lingfeng Bao, and Wenchao Xu for their help in the collection of status histories of bug reports that we analyze in this study.

References

Anvik J, Murphy GC (2011) Reducing the effort of bug report triage: recommenders for development-oriented decisions. TOSEM 20(3):10
Anvik J, Hiew L, Murphy GC (2005) Coping with an open bug repository. In: ETX, pp 35–39
Bhattacharya P, Neamtiu I, Shelton CR (2012) Automated, highly-accurate, bug assignment using machine learning and tossing graphs. J Syst Softw 85(10):2275–2292
Cohen WW (1995) Fast effective rule induction. In: ICML
Crammer K, Singer Y (2001) On the algorithmic implementation of multiclass kernel-based vector machines. J Mach Learn Res 2
Cubranic D, Murphy GC (2004) Automatic bug triage using text categorization. In: SEKE, pp 92–97
Duda R, Hart P, Stork D (2000) Pattern classification. Wiley Interscience
Eclipse (2012) http://wiki.eclipse.org/Bug_Reporting_FAQ#What_is_the_difference_between_Severity_and_Priority.3F
Forman G (2008) Bns feature scaling: an improved representation over tf-idf for svm text classification. In: CIKM
Gegick M, Rotella P, Xie T (2010) Identifying security bug reports via text mining: an industrial case study. In: MSR
Han J, Kamber M, Pei J (2011) Data mining: concepts and techniques. Morgan Kaufmann
Hiew L (2006) Assisted detection of duplicate bug reports. Master's thesis, The University Of British Columbia
Hosseini H, Nguyen R, Godfrey M (2012) A market-based bug allocation mechanism using predictive bug lifetimes. In: CSMR
Huang L, Ng V, Persing I, Geng R, Bai X, Tian J (2011) AutoODC: automated generation of orthogonal defect classifications. In: ASE
Jalbert N, Weimer W (2008) Automated duplicate detection for bug tracking systems. In: DSN
Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: ESEC/SIGSOFT FSE, pp 111–120
Khomh F, Chan B, Zou Y, Hassan AE (2011) An entropy evaluation approach for triaging field crashes: a case study of mozilla firefox. In: WCRE
Kim S, Whitehead EJ (2006) How long did it take to fix bugs? In: MSR
Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: MSR
Lamkanfi A, Demeyer S, Soetens Q, Verdonck T (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: CSMR
Manning CD, Raghavan P, Schutze H (2008) Introduction to information retrieval. Cambridge
Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: ICSM
Nguyen AT, Nguyen TT, Nguyen TN, Lo D, Sun C (2012) Duplicate bug report detection with a combination of information retrieval and topic modeling. In: ASE
PorterStemmer (2011) www.ils.unc.edu/~keyeg/java/porter/PorterStemmer.java
Robertson S, Zaragoza H, Taylor M (2004) Simple BM25 extension to multiple weighted fields. In: CIKM
Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: ICSE, pp 499–510
Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: ICSE
Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: ASE
SVM-MultiClass (2011) http://svmlight.joachims.org/svm_multiclass.html
Tamrawi A, Nguyen TT, Al-Kofahim J, Nguyen TN (2011) Fuzzy set-based automatic bug triaging. In: ICSE, pp 884–887
Tian Y, Lo D, Sun C (2012) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: WCRE
Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: ICSE, pp 461–470

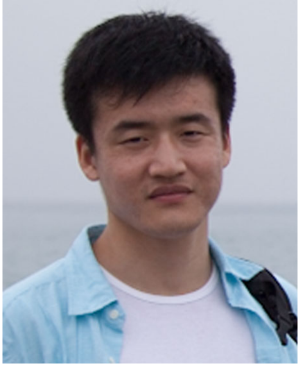
Weiß C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: MSR, p 1
WEKA (2011) http://www.cs.waikato.ac.nz/ml/weka/. Weka 3: Data Mining Software
Xia X, Lo D, Wen M, Shihab E, Zhou B (2014) An empirical study of bug report field reassignment. In: CSMR-WCRE



Yuan Tian is currently a PhD student in the School of Information Systems, Singapore Management University. She started her PhD program in 2012. Previously, she received her bachelor degree in the College of Computer Science and Technology from Zhejiang University, China in 2012. Her research is in software system and data mining area. Particularly, she is interested in analyzing textual information in software repositories.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He works in the areas of software engineering and data mining. He is particularly interest in specification mining, debugging, software text analytics, frequent pattern mining, and social network mining. He is a member of the Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Xin Xia received his PhD degree from the College of Computer Science and Technology, Zhejiang University, China in 2014. He is currently a Research Scientist in the College of Computer Science and Technology, Zhejiang University. His research interests include software analytic, empirical study, and mining software repository. He is a member of the Institute of Electrical and Electronics Engineers.



Chengnian Sun is a Postdoctoral Scholar at Department of Computer Science, University of California, Davis, USA. His research interests include compiler testing, software debugging, and mining software repositories. He received his bachelor degree of engineering from Northeastern University China in 2007, and his PhD degree in computer science from School of Computing, National University of Singapore in 2013.