DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis

Yuan Tian[1], David Lo[1], and Chengnian Sun[2]
[1]Singapore Management University, Singapore
[2]National University of Singapore, Singapore
{yuan.tian.2012,davidlo}@smu.edu.sg, suncn@comp.nus.edu.sg

***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 58.61%.**

I. 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).

Although bug reporting could potentially improve software quality, the number of such reports could be too many for developers to handle. In 2005, it is reported that "everyday almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [2]. Thus developers often need to prioritize which bugs are to be given attention first. In Bugzilla there are 5 priority levels: P1, P2, P3, P4, and P5. P1 is the highest priority and often the software system can only be shipped if these high priority bugs are fixed. P5 on the other hand is the lowest priority and bugs assigned this priority might remain unfixed for a long period of time.

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 (PreD̲icting PR̲iority via Multi-Faceted FactO̲r AN̲alysE̲s) to aid triagers in assigning priority labels to bug reports. Inside DRONE, we include our new classification engine named GRAY (ThresholdinG̲ and Linear R̲egression to ClA̲ssifY̲ Imbalanced Data) 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

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.31

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 [16], Lamkanfi et al. [13], [14], and our own previous work [27]. 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 [16] for bug priority prediction. Our experiments demonstrate that we can achieve 58.61% 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 introduce a new machine learning framework, named DRONE, that would consider these factors and predict the priority of a bug given its report. We also 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 58.61%.

The structure of this paper is as follows. In Section II, we describe preliminary information on bug report, text pre-processing, and measuring similarity of bug reports. In Section III, we describe our proposed approach. Section IV presents the result of our experiments. Next, we discuss interesting issues in Section V. Related work is presented in Section VI. Finally, we conclude and discuss future work in Section VII.

II. Preliminaries & Problem Definition

In this section, we first describe bug reports and bug report reporting process. Next, we present an approach to pre-process textual documents. Then, we highlight REP [22], which is a recently proposed state-of-the-art similarity measure of bug reports. Finally, we present our problem definition.

A. Bug Reports and Reporting Process

Developers often desire feedback on defects that exist on 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 I.

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 assign an appropriate priority level. 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 II. Note that bug reports shown in the same box (e.g., 4629 and 4664) are duplicates of one another. After assigning a priority level to the bug report, 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.

TABLE I
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.

TABLE II
EXAMPLES OF BUG REPORTS FROM ECLIPSE

	ID	Summary	Product	Component	Severity	Priority
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 to file option (1GKXC30)	Platform	SWT	normal	P3
	5126	StyledText printing should implement "print to file"	Platform	SWT	normal	P3

B. 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 affect 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 appear 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 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 [21] to process the text as it is commonly used by many prior studies, e.g., [16], [13], [14], [29].

C. 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 [9], Runeson et al. [19], Wang et al. [29], etc.

The most recent approach proposed to measure the similarity of bug reports is REP which is proposed by Sun et al. [22]. Their approach extends BM25F [18] 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

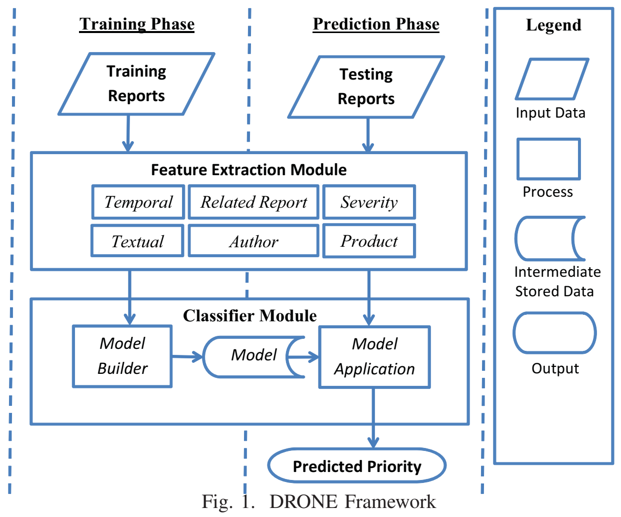

Fig. 1. DRONE Framework

their similarity.

D. Problem Definition

"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."

III. 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.

A. Overall Framework

Our framework, named DRONE (PreDicting PRiority via Multi-Faceted FactOr ANalysEs), is illustrated in Figure 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 affect 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.

B. 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 `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 III.

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 II. Each of the resultant word token 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.

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. [22] – described in Section II. 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

TABLE III
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.

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 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.

C. 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 (ThresholdinG and Linear Regression to ClAssifY Imbalanced Data). We illustrate our classification engine in Figure 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. This is so as 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.

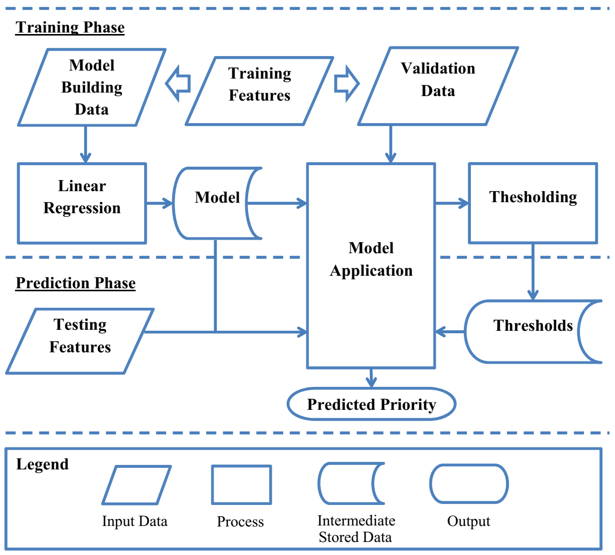

Fig. 2. GRAY Classification Engine

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 $T$
```

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.

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 split the training data into two (by default, a 50-50 split): model building and validation. 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.

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).

IV. Experiments & Analysis

In this section, we first describe the 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 followed by our findings.

A. Dataset

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 by and used by many developers around the world. We consider the bug reports submitted from October 2001 to December 2007 and download them from Bugzilla[1]. We

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

TABLE IV
DATASET DETAILS

Dataset	Period		REP$^-$ Training Reports		DRONE Training Reports	Testing Reports
	From	To	#Duplicate	#All	#All	#All
Eclipse	2001-10-10	2007-12-14	200	3,312	87,649	87,648

collect only defect reports and ignore those that correspond to feature requests.

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. [22]). 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 [7], [17], [19]. We show the distribution of bug reports used for training and testing in Table IV.

B. Experimental Setting

We compare our approach with an adapted version of Severis which was proposed by Menzies and Marcus [16]. 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.

C. 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 [16]. The definitions of precision, recall, and F-measure for a priority level P are given below:

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

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

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

D. Research Questions

RQ1 How accurate is our proposed approach as compared with the baseline approaches namely $Severis^{Prio}$ and $Severis^{Prio+}$?

RQ2 How efficient is our proposed approach as compared with the baseline approaches namely $Severis^{Prio}$ and $Severis^{Prio+}$?

RQ3 Which of the features are the most effective in discriminating the priority levels?

RQ4 What are the effectiveness of various classification algorithms in comparison with GRAY in predicting the priority levels of bug reports?

E. Experimental Results

Here, we present the answers to the four research questions. The first two compare DRONE with Severis^{Prio} and $Severis^{Prio+}$ on two dimensions: accuracy and efficiency. The best approach must be accurate and yet can complete training and prediction fast. 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.

RQ1: Accuracy of DRONE vs. Accuracy of Baselines

The result of DRONE is shown in Table V. We note that we can predict the P1, P2, P3, P4, and P5 priority levels by an F measure of 41.76%, 11.64%, 86.85%, 0.43%, and 8.01% respectively. The F-measures are better for P1, P2, and P3 priority levels but are worse for P4, and P5 priority levels. We believe in report prioritization high accuracy for high priority bugs is much more important than high accuracy for low priority bugs.

TABLE V
PRECISION, RECALL, AND F-MEASURE FOR DRONE

Priority	Precision	Recall	F-Measure
P1	41.15%	42.39%	41.76%
P2	10.92%	12.46%	11.64%
P3	91.36%	82.77%	86.85%
P4	0.24%	1.77%	0.43%
P5	4.97%	20.72%	8.01%
Average	29.73%	32.02%	29.74%

The result for Severis^{Prio} is shown in Table VI. We note that Severis^{Prio} can predict the P1, P2, P3, P4, and P5 priority levels by an F-measure of 0.00%, 0.00%, 93.76%, 0.00%, and 0.00% respectively. 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 V), we note that we can improve the average of the F measures by a relative

TABLE VI
PRECISION, RECALL, AND F MEASURE FOR SEVERIS^{Prio}

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 VII
PRECISION, RECALL, AND F MEASURE FOR SEVERIS^{Prio+}

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%

improvement of 58.61% (i.e., $(29.74 - 18.75)/18.75 \times 100\%$). Thus, clearly DRONE performs better than Severis^{Prio}.

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

RQ2: Efficiency of DRONE vs. Efficiency of Baselines

We compare the runtime of DRONE with those of Severis^{Prio} and Severis^{Prio+}. The result is shown in Table VIII. The four columns refer to the average feature extraction time (for training data), the model building time, the average feature extraction time (for testing data), and the average model application time. We could note that the time for feature extraction is slower for DRONE than the two variants of Severis. This is the case as DRONE utilizes more features than the two variants of Severis. Severis^{Prio} only utilizes the textual features of bug reports. Severis^{Prio+} only utilizes the textual and severity features of bug reports. The time for model building however is faster for DRONE than the two variants of Severis. We compare the efficiency of the approaches since the required running time determines the usability of the system for triagers.

TABLE VIII
EFFICIENCY OF SEVERIS^{Prio}, SEVERIS^{Prio+}, AND DRONE. FE = AVERAGE FEATURE EXTRACTION TIME. MB = MODEL BUILDING TIME. MA = AVERAGE MODEL APPLICATION TIME.

Approach	Time (in seconds)			
	FE (Train)	MB	FE (Test)	MA
Severis^{Prio}	<0.01	812.18	<0.01	<0.01
Severis^{Prio+}	<0.01	773.62	<0.01	<0.01
DRONE	0.01	69.25	0.02	<0.01

RQ3: 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 [15] and Fisher score [5] 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).

TABLE IX
TOP-10 FEATURES IN TERMS OF FISHER SCORE

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

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 IX 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.

We notice that the 10th 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.

RQ4: 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 [4], RIPPER [3], and Naive Bayes Multinomial [15]) and compare their F-measures across the five priority levels with GRAY. We use the implementation of SVM-MultiClass available from [24]. We use the implementations of RIPPER and Naive Bayes Multinomial in WEKA [1]. We show the result in Table X. We notice that in terms of average F-Measures GRAY outperforms SVM-MultiClass by a relative improvement of 58.61%. 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.

V. 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

TABLE X
COMPARISONS OF AVERAGE F-MEASURES OF GRAY VERSUS OTHER CLASSIFIERS. 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).

Class.	**F-Measures**					
	P1	P2	P3	P4	P5	Ave.
GRAY	41.76%	11.64%	86.85%	0.43%	8.01%	29.74%
SM	0%	0%	93.76%	0%	0%	18.75%
RIPPER	CC	CC	CC	CC	CC	CC
NBM	OOM	OOM	OOM	OOM	OOM	OOM

with a trivial approach that simply assigns a bug report to the most common priority level.

A. Threats to Validity

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 [16]. 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. Threats of 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.

B. Dismal Performance of Baseline

Results of Severis^{Prio} and Severis^{Prio+} shown in Table VI and Table VII 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 "thinks" 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.

C. 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. The trivial approach simply labels every bug as P3 based on the fact that nearly 90% of the bug reports are assigned P3 priority level. The result of this trivial approach is the same as those of Severis^{Prio} and Severis^{Prio+} shown in Tables VI and VII respectively. This means that this trivial approach can predict the P1, P2, P3, P4 and P5 priority levels by an F-measure of 0.00%, 0.00%, 93.76%, 0.00% and 0.00% respectively, with an average F-measure of 18.75%. Thus our model 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.

VI. RELATED WORK

Menzies and Marcus are the first to predict the severity of bug reports [16]. 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 [3]. 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 [13]. 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 [14]. 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 [27]. 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 [20].

Khomh et al. automatically assign priorities to Firefox crash reports in Mozilla Socorro server based on the frequency and entropy of the crashes [11]. 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.

There are other lines of work that also analyze bug reports; these include the series of work on duplicate bug report detection [19], [23], [22], [28], bug localization [31], bug categorization [6], [8], [26], bug fix time prediction [12], [30], and bug fixer recommendation [10], [25]. Our work is also orthogonal to these studies.

VII. Conclusion and Future Work

In this work, we propose a framework named DRONE (PreDicting PRiority via Multi-Faceted FactOrs ANalysEs) 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 (ThresholdinG and Linear Regression to ClAssifY Imbalanced Data) 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 [16]. 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 58.61%.

In the future, 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. For instance, we can try to construct a linear regression model using only the most discriminative features and evaluate the resulting solution. We also plan to analyze the impact of inaccuracies in the thresholding process on the final result of DRONE.

References

[1] "http://www.cs.waikato.ac.nz/ml/weka/," Weka 3: Data Mining Software.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, 2005.

[3] W. W. Cohen, "Fast effective rule induction," in *ICML*, 1995.

[4] K. Crammer and Y. Singer, "On the algorithmic implementation of multiclass kernel-based vector machines," *Journal of Machine Learning Research*, vol. 2, 2001.

[5] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley Interscience, 2000.

[6] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *MSR*, 2010.

[7] L. Hiew, "Assisted detection of duplicate bug reports," Ph.D. dissertation, The University Of British Columbia, 2006.

[8] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *ASE*, 2011.

[9] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *DSN*, 2008.

[10] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *ESEC/SIGSOFT FSE*, 2009.

[11] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *WCRE*, 2011.

[12] S. Kim and E. J. W. Jr., "How long did it take to fix bugs?" in *MSR*, 2006.

[13] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010.

[14] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, 2011.

[15] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge, 2008.

[16] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008.

[17] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *ASE*, 2012.

[18] S. Robertson, H. Zaragoza, and M. Taylor, "Simple BM25 Extension to Multiple Weighted Fields," in *CIKM*, 2004.

[19] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007.

[20] http://wiki.eclipse.org/Bug_Reporting_FAQ#What_is_the_difference_between_Severity_and_Priority.3F.

[21] www.ils.unc.edu/~keyeg/java/porter/PorterStemmer.java.

[22] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[23] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.

[24] http://svmlight.joachims.org/svm_multiclass.html.

[25] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging," in *ICSE*, 2011.

[26] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCRE*, 2012.

[27] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *WCRE*, 2012.

[28] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *CSMR*, 2012.

[29] 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 *ICSE*, 2008.

[30] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *MSR*, 2007.

[31] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, 2012.