Predicting Bug Report Fields Using Stack Traces and Categorical Attributes

Korosh K. Sabor
Concordia University
Montréal, QC, Canada
k_kooche@ece.concordia.ca

Abdelwahab Hamou-Lhadj
Concordia University
Montréal, QC, Canada
abdelw@ece.concordia.ca

Abdelaziz Trabelsi
Concordia University
Montréal, QC, Canada
trabelsi@ece.concordia.ca

Jameleddine Hassine
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa

ABSTRACT

Studies have shown that the lack of information about a bug often delays the bug report (BR) resolution process. Existing approaches rely mainly on BR descriptions as the main features for predicting BR fields. BR descriptions, however, tend to be informal and not always reliable. In this study, we show that the use of stack traces, a more formal source, and categorical features of BRs provides better accuracy than BR descriptions. We focus on the prediction of faulty components and products, two important BR fields, often used by developers to investigate a bug. Our method relies on mining historical BRs in order to predict faulty components and products of new incoming bugs. We map stack traces of historical BRs to feature vectors, weighted using TF-IDF. The vectors, together with a selected set of BR categorical information, are then fed to a classification algorithm. The method also tackles the problem of unbalanced data. Our approach achieves an average accuracy of 58% (when predicting faulty components) and 60% (when predicting faulty products) on Eclipse dataset and 70% (when predicting faulty components) and 70% (when predicting faulty products) on Gnome dataset. For both datasets, our approach improves over the method that uses BR descriptions by a large margin, up to an average of 46%.

KEYWORDS

Software Bugs Reports, Mining Software Repositories, Software Maintenance and Evolution, Machine Learning

1 Introduction

Bug report tracking systems are designed to help users and developers report bugs. By using these systems, end users submit a bug reports (BRs) by entering a description, attaching a stack trace, and providing categorical attributes such as severity level, platform information, and the faulty products and components. When a BR is submitted, it is examined by a triaging team with the objective of redirecting it to the developers who are familiar with the affected software components in order to provide a fix. To this end, triagers rely heavily on the information provided in the BRs. The problem is that this information is not always reliable. It has been shown that it is common for users to enter incorrect BR fields [2, 7]. Bettenburg et al. [2] showed that there is an important gap between what users provide as input and what developers need to fix a bug. They added that since end users do not usually have technical knowledge about the system, it is very difficult for them to report BR fields accurately. In addition, Xia et al. [20] showed that 80% of BRs have their fields reassigned several times after they have already been submitted to developers.

Among the reassigned BR fields, the component and product fields are the ones that tend to be reassigned the most [6, 7, 12 21]. These fields also happen to be very important since they are used by triaging teams to route BRs to the right development team as shown by Somasundaram et al. [17]. The same authors empirically showed that incorrect components often delay the resolution of BRs.

Who	When	What	Removed	Added
remy.suen	2008-01-17 14:05:31 EST	CC		remy.suen
pwebster	2008-01-17 15:04:14 EST	Assignee	Platform-UI-Inbox	php.ui-inbox
		Component	UI	PHP Explorer View
		Product	Platform	PDT
		Version	3.4	unspecified
spektom	2009-05-08 15:30:51 EDT	CC		spektom
		Component	PHP Explorer & Projects management	Common
		Product	PDT	DLTK
		Version	unspecified	1.0
spektom	2009-05-08 15:35:34 EDT	Status	NEW	RESOLVED
		Resolution	---	FIXED

Figure 1: Eclipse BR #215679 history information (from https://bugs.eclipse.org/bugs/show_activity.cgi?id=215679)

As a motivating example, consider the history of Eclipse BR #215679, shown in Figure 1. The report was first assigned to a developer to be fixed on January 17th, 2008. The component and product fields were first changed from UI and Platform to PHP Explorer View and PDT, respectively. After 16 months, these fields were changed again to Common and DLTK, which were the correct fields. Clearly there is a need to develop techniques and tools that

CASCON'19, November 4-6, 2019, Toronto, Canada

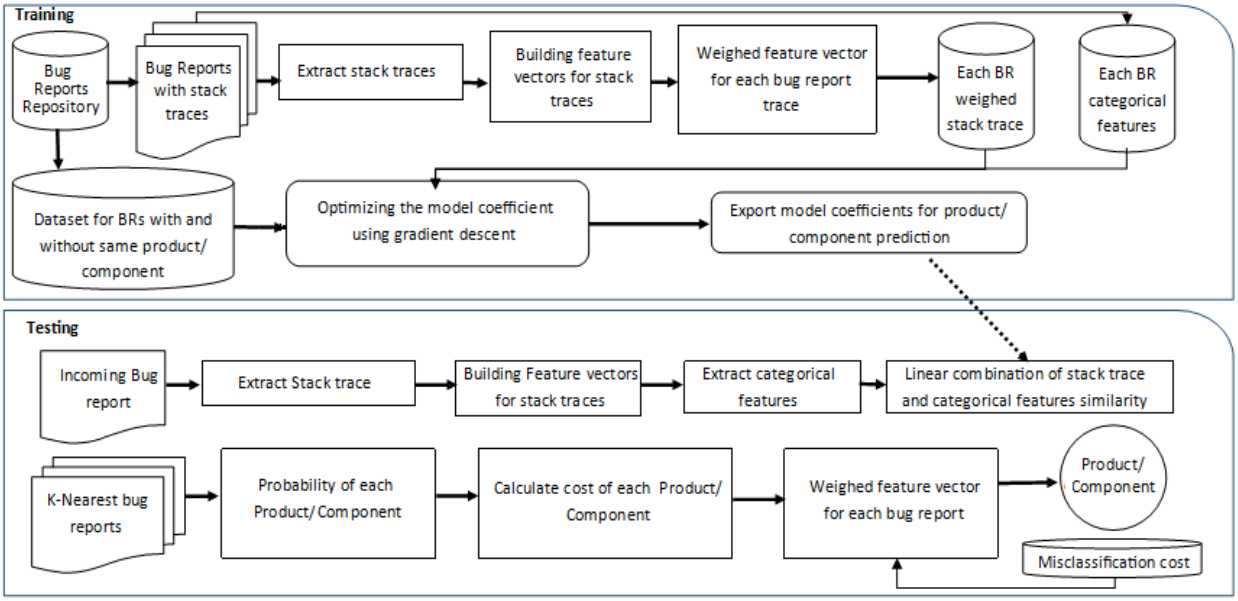

Figure 2: Overall approach

can predict the correct product and component fields at the time of submission of a BR. Such techniques can provide tremendous help to triagers in processing the BRs.

There exist studies that aim at predicting these fields. Sureka et al. [12] proposed an approach based on BR descriptions to predict faulty components. Wang et al. [19] compared the effectiveness of different machine learning techniques to predict faulty components. These techniques treat the problem as a classification problem by building a model from historical BRs that can later be used to predict whether or not the field of an incoming BR will be reassigned (and ideally predict the correct field), relying on BR descriptions as the main features. BR descriptions, however, vary in the quality of their content as shown by Bettenburg et al. [2]. In this paper, we propose an alternative approach to automatically predict the component and product fields of BRs using a combination of stack traces and categorical attributes (more precisely, the system version, severity, and platform). A stack trace contains a sequence of running functions and threads in the system at the time of the crash. They have been used to diagnose the causes of failures [28], bug reproduction [29, 30], and prediction of BR severity [13]. This is because they tend to be a more formal source of information than BR descriptions and comments that are entered by end users (including developers) using natural language. Stack traces are therefore a useful alternative, especially when BR descriptions and comments suffer from quality problems due to noise in the data and the ambiguity and imprecision associated with the use of natural language.

The motivation behind using categorical features comes from the work of Xia et al. [20] who showed that these features (called meta features in their paper) enhance the prediction of BR fields that will most likely be reassigned. Categorical features are also used by Sun et al. in [16] as additional features to detect duplicate BRs.

We show the effectiveness of our approach by applying it to Eclipse and Gnome BRs. Moreover, we show that our approach outperforms techniques that rely solely on BR descriptions (e.g., [12, 19]). To the best of our knowledge, this is the first time that stack traces are used with or without other features to predict faulty product and component fields of BRs.

The remainder of this paper is organized as follows. Our approach is described in Section 2. The experimental protocol is provided in Section 3. We present and discuss the results in Section 4. In Section 5, we discuss the threats to validity of our approach. We present related work in Section 6, followed by a conclusion and future work.

2 Approach

Figure 2 shows our approach for predicting the correct products and components of BRs, which is composed of two main phases: training and testing.

For this study, we train our classification algorithm using a linear combination of stack trace similarity and BR categorical attributes, namely the system version, severity, and platform. We chose these categorical attributes because they are the main BR fields that describe the properties of the faulty system. We present the steps of the training and testing phases of our approach in the following subsections.

2.1 Training Phase

We build a training model using the distinct functions extracted from all stack traces of the BR training dataset. Then, for each stack trace T_i , a feature vector is constructed and weighted using TF-IDF (term frequency/inverse document frequency). More formally, each stack trace T is mapped to a vector of size m functions, where each function name $f_i \in \Sigma$ in the vector is either one (appeared in the stack trace) or zero (did not appear in the stack trace). The feature vector is weighted by the term frequency (tf):

$$\phi_{tf}(f, T) = freq(f_i); \quad i = 1, \dots, m \tag{1}$$

In Equation (1), $freq(f_i)$ is the number of times the function f_i appears in Tdivided by the total number of functions in the stack trace, L. We use IDF to give weight to rare functions while decreasing the weight of frequent functions. The feature vector weighed by the TF-IDF is therefore given by:

$$\phi_{tf.idf}(f,T,\Gamma) = \frac{\mathrm{K}}{df(f_i)} freq(f_i); \quad i = 1,\dots,m \qquad (2)$$

where the document frequency $df(f_i)$ is the number of stack traces T_k in the collection of Γ of size K which contains function name f_i.

To compare two stack traces, we measure the distance between their corresponding feature vectors using the cosine similarity measure. (Other distance metrics can also be used). Given V1 and V2, two features vectors representing stack traces from two BRs, the cosine similarity is as follows [20]:

$$Cos(\theta) = \frac{V_1.V_2}{|V_1|.|V_2|} \qquad (3)$$

To add categorical attributes, we measure the similarity of two BRs B_1 and B_2 as follows:

$$SIM\ (B_1,B_2) = \sum_{i=1}^{4} w_i * feature_i \qquad (4)$$

In Equation (4), the parameters $feature_1$, $feature_2$, $feature_3$, and $feature_4$ are defined in the Table 1.

The SIM function in Equation (4) contains four parameters (w_1, w_2, w_3, w_4) that we used to weigh each feature. These weights are adjusted in a separate training phase. We use 10% of our dataset to train these parameters. To optimize parameters, we use the Rank Net Cost function (RNC) provided by Sun et al. [16], which is defined in Equation (5). Note that other optimization functions can be used. The comparison of various optimization functions is out of the scope of this paper.

$$Y = SIM(irr,q) - SIM\ (rel,q)$$

$$RNC(Y) = Log(1+e^{Y}) \qquad (5)$$

Table 1. Features used to measure BR similarities

Feature	Value
$feature_1$	Cosine similarity of stack traces based on the constructed term vectors.
$feature_2$	$= \begin{cases} 1 & if\ B1.version = B2.version \\ 0 & Otherwise \end{cases}$
$feature_3$	$= \begin{cases} 1 & if\ B1.severity = B2.severity \\ 0 & Otherwise \end{cases}$
$feature_4$	$= \begin{cases} 1 & if\ B1.platform = B2.platform \\ 0 & Otherwise \end{cases}$

The values of w_1, w_2, w_3, w_4 are obtained by minimizing the cost function of Equation (5), which is achieved by maximizing the similarity of BRs with the same product or component fields and minimizing the similarity of BR with different products or components. To achieve this, we use the gradient descent algorithm, provided by Sun et al. [16].

The algorithm adjusts each free parameter x in each iteration according to the value of the coefficient Ƞ and the partial derivative of RNC with respect to each free parameter x. Then the four free parameters (w_1, w_2, w_3, w_4) are used to calculate the similarity of each incoming BR to all previous BRs in the dataset to predict the faulty product and component fields using the cost sensitive K-Nearest Neighbor (KNN).

2.2 Testing Phase

We simulate the submission of BRs to a bug tracker based on real data. Each time a new BR is submitted we will compare it to previous BRs by measuring the similarity between its stack trace (more precisely the corresponding feature vector) and categorial attributes to all the previous BRs that were submitted. In other words, only stack traces and categorical features of BRs submitted before the current simulated time can be used. Note that this requires updating the existing feature vectors by adding the functions that were not seen before as well as updating the TF-IDF weights. This technique was also used by Lerch et al. when detecting duplicate bug reports [26].

The calculated similarities are then used to build a list of similarity values that shows how similar the current stack trace of the incoming BR is to all previous stack traces of all BRs in the training dataset. We use the cost sensitive KNN algorithm to retrieve the most similar BRs to the incoming BR.

The KNN classification process is performed into two phases. In the first phase, the similarity of the incoming BR (B_i), in the testing dataset to all the BRs in the training dataset is calculated. In the second phase, the K-nearest BRs in the training dataset are then selected and the label of the incoming B_i is selected by a majority vote. That is, the label of the instance X associated to B_i is determined using a labelled dataset C and a majority voting scheme according to the following equation [11]:

$$C(X) = \underset{c_j \in C}{argmax}\ score\ (c_j, neighbors_k(X)) \qquad (6)$$

In the above equation, $neighbors_k(X)$ is the K-nearest neighbors of instance X, argmax returns the label which maximizes the score function; it is defined as follows [11]:

$$Score(c_j,N) = \sum_{Y \in N} [class(y) = c_j] \qquad (7)$$

In the above equation, $class(y) = c_j$ is evaluated to one if $class(y) = c_j$ and to zero if $class(y) \neq c_j$. That is, the label with the highest frequency of occurrences among the K-returned labels is considered as the output label.

To further improve the prediction capability of our approach, more weights have been given to BRs of the training dataset closer to the incoming BR, B_i. We achieve this using the reciprocal of the similarity of bugs which is the outcome of each of the four free parameters involved in Equation (4). More formally, let the distance of the closest BR in the sorted list of the K-nearest instances be $dist_1$, and the distance of the farthest bug be $dist_k$. The weight of each label can be computed by the following equation introduced by Gou et al. [24]. In this equation, $dist_i$ is the distance of BR i.

$$w_i = \begin{cases} \frac{dist_k - dist_i}{dist_k - dist_1} & if\ dist_k \neq dist_1 \\ 1 & if\ dist_k = dist_1 \end{cases} \qquad (8)$$

It follows that the score function defined by Equation (7) must be updated to incorporate the calculated weights in Equation (8) [11]. The following equation encompasses the desired changes.

$$\text{Score}\,(c_j, N) = \sum_{Y \in N} w(x,y) \times [class(y) = c_j] \quad (9)$$

In the above equation, $w(x,y)$ is the weight of each instance in the top K-similar returned instances, which is obtained from its distance to the incoming B_i and the associated instance x. Then, the label with the resulting high score is selected to be the output label.

In large software repositories such as Eclipse, Gnome, and Mozilla, some products or components have fewer BRs in the bug tracking system, which results in an unbalanced distribution of labels, causing a bias towards the majority class labels. We used cost sensitive learning to overcome the unbalanced dataset distribution problem with Eclipse and Gnome datasets. To apply cost sensitive learning, we first convert the output of the KNN classifier into the probability of the test instance belonging to each class label. Then, we construct a cost matrix in which the probability of belonging to each class label is exchanged with an average cost of belonging to each class label. If we consider number of different class labels as C, the number of instances of class j in the training set as s_j, and the number of instances of the majority class in the training set as s then the misclassification cost of each class label C_j could be calculated by Equation (10) [13].

$$MC_j = \frac{s}{s_j} \quad (10)$$

Assume we have M classes and the incoming bug belongs to each of these classes with probabilities of $P_1 \ldots\ldots P_m$, and assume that each class has a misclassification cost of $CO_1 \ldots\ldots CO_m$, then the cost of assigning the bug report to each of those classes is calculated by Equation (11) [23].

$$CCO_i = \sum_{j \in m \text{ and } j \neq i} CO_j \times P_j \quad (11)$$

Based on the output probabilities and using a cost matrix, the classifier makes an optimal cost-sensitive prediction, by choosing the class label with the least classification cost.

3 Experimental Protocol

We conducted experiments to address the following research questions:

RQ1. Can stack traces and categorical features (system version, severity, and platform) be used to predict the product field of a BR? If so, what would be the prediction accuracy and how does it compare with the use of BR descriptions?

RQ2. Can stack traces and categorical features (system version, severity, and platform) be used to predict the component field of a BR? If so, what would be the prediction accuracy and how does it compare with the use of BR descriptions?

RQ3. How does our approach compare to a random classifier?

3.1 Datasets

In this work, we used BRs extracted from two large open-source software projects: Eclipse and Gnome. These systems have their BRs open and accessible to researchers and have been widely used in the literature [1, 8, 9, 18, 22, 25]. We included Eclipse BRs submitted between October 2001 to February 2015 and all Gnome BRs for the period of February 1999 to August 2015.

In this work, we focused only on Eclipse products and components that pertain to Eclipse core and did not include the plugins. Eclipse has five products: Platform, JDT, PDE, Equinox and E4, each of which contains a set of components.

Gnome is a collection of Unix-based projects. Although it uses the same bug tracking system as Eclipse, Gnome's BRs are structured slightly differently. The concept of product in Gnome refers to a BR class; each class contains a set of components (called products in Gnome).

In Eclipse and Gnome bug repositories, stack traces are embedded in BR descriptions. We used regular expressions to extract stack traces from bug report descriptions. For Eclipse we used the same regular expression provided by Lerch et al. [26]. For Gnome, after examining carefully the way stack traces are organized, we designed the following regular expression. In Gnome, a stack trace starts with a frame number following by a hex number. Function name and the parameters are presented next. Based on the debugging configuration, function parameters are followed by keywords 'from' or 'at', and library or filename.

```
([#NUMBER] [HEX ADDRESS] [IN] [FUNCTION NAME] [(] [PARAMETERS] [)] ([FROM] | [AT]) ([LIBRARYNAME] | [FILENAME]))*
```

Figure 3: Regular expression for extracting stack traces from Gnome BR descriptions

The total number of Eclipse BRs is 193,177, but only 19,458 (10%) have stack traces. This low percentage results from the fact that up to 2015, stack traces had to be appended manually by users. Automatic submission of stack traces to the Eclipse BR repository has been made possible by Eclipse at the end of 2015. So far, these stack traces have not been made publicly available. In the case of Gnome, the total number of BRs is 629,549, among which 201,580 (32%) have stack traces. The sparsity of stack traces is the main limitation of our approach. Nevertheless, we believe that it is still important to investigate the use of stack traces, especially that there is a recognized need to have stack traces for debugging, bug reproduction, and other software maintenance tasks. We should expect to see more bug reporting systems collect stack traces automatically whenever a BR is submitted.

3.2 Predicting BR product and component fields using BR description

We compared our approach to an approach that uses the description of BRs such as the one presented by Sureka in [12]. The authors applied two different techniques, TF-IDF and a dynamic language model classifier, to predict faulty components. They showed that these approaches have similar accuracy. Since both approaches have the same performance, we compared our approach which uses stack traces and categorical features to Sureka's approach using TF-IDF. Sureka did not tackle the unbalanced label distribution problem, and because we are applying their approach to an unbalance dataset, we have added the cost sensitive KNN to their approach.

We extracted the descriptions from all BRs in our datasets. We tokenized words in the descriptions and built feature vectors, which

consist of all distinct words in the descriptions of a BR. Next, we used TF-IDF (Equation 2) to give weight the feature vector of each BR. We created a sorted set of BRs using the submission date. With the incoming of each new BR, its weighed feature vector is compared to the weighted feature vector of all previous BRs in the sorted set. Next, we used the same cost sensitive K nearest neighbor method (Equation 11) to select the BR that is most similar.

3.3 Predicting BR product and component fields using a random classifier

Our proposed classification approach is also compared to a method that predicts the faulty product and component of bugs with respect to different class labels. Assume we have N faulty product classes and the number of BRs that belong to each class is $B_{p_1} \dots \dots B_{p_N}$. The accuracy of correctly predicting a faulty product of each bug by randomly choosing product label P_i is calculated by Equation (12). We use the same technique for components:

$$Accuracy\ (P_i) = \frac{B_{p_i}}{\sum_{j=1}^{N} B_{p_j}} \quad (12)$$

3.4 Evaluation Metrics

We used precision, recall, and F-measure to assess the effectiveness of our approach. These metrics are widely used in the literature [7, 20] to evaluate the accuracy of a classifier. We defined precision as Equation (13) We calculated the precision for each product and component label separately.

$$Precision\ (p_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label p_L}{\#\ of bugs\ predicted\ to\ have\ label p_L} \quad (13)$$

Recall is defined as (14).

$$Recall\ (P_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ P_L}{\#\ of\ bugs\ actually\ having\ label\ P_L} \quad (14)$$

We built the confusion matrix separately for each product or component field. We combined precision and recall values and present them as one value, F-measure:

$$F - measure\ (p_L) = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (15)$$

To compare the results of our approach to an approach that uses BR descriptions, we measured the improvement achieved by one method over the other. More precisely, if we denote the F-measure of Sureka's [12] approach as $F\_measure_T$ and the F-measure of our approach by $F\_measure_{SC}$, we calculate the improvement using Equation (16) as follows:

$$\text{Improvement} = \frac{F-measure_{SC} - F-measure_T}{F-measure_T} \quad (16)$$

Because we have a large number of components for each product and that the precision and recall must be calculated separately for each component, we used the macro-average precision to show the average precision of the components of each product. If we denote precision of the first component as P_{C1} and the n'th component precision as P_{Cn}, we can use the equation provided by Manning et al. [10] to calculate the macro-average precision:

$$Macro_{average} precision = \frac{P_{C1} + P_{C2} + \cdots + P_{Cn}}{n} \quad (17)$$

Similarly, if we denote recall of the first component as R_{C1} and the n'th component recall as R_{Cn}, the macro-average recall is calculated using the following equation [10].

$$Macro_{average} recall = \frac{R_{C1} + R_{C2} + \cdots + R_{Cn}}{n} \quad (18)$$

4 Results and Discussion

4.1 Results

For simplicity reason, we use the notation BRFPst+cat to refer to our approach for predicting BR fields using stack traces and categorical features. We also use BRFPdesc to refer to an approach that uses BR descriptions.

Table 2 and Table 4 show the precision, recall, and F-measure of BRFPst+cat in Eclipse and Gnome datasets. In our experiments, we varied K from 1 to 10 and recorded K which provides the best accuracy. This is a common practice in machine learning when using KNN. Table 3 and Table 5 show the macro precision, recall and F-measure of our approach when predicting the BR component fields in Eclipse and Gnome datasets. Note that the detailed precision and recall for each component can be found on http://www.ece.concordia.ca/~abdelw/cascon19.

RQ1. Can stack traces and categorical features be used to predict the <u>product</u> field of a BR and if so, what would be the accuracy and how does it compare to the use of BR descriptions?

When applied to Eclipse products, the results show that our approach, BRFPst+cat, predicts faulty products with an average F-measure of 60%. The average precision and recall is 58% and 62%, respectively. For Gnome, we applied our approach to seven products and predicted faulty products with an average precision of 74% and an average recall of 67%. The average F-measure is 70%.

Table 2 shows the average F-measure improvement of predicting faulty products using our approach for Eclipse compared to the use of BR descriptions, BRFPdesc. Based on the results of Table 2, we have improved the average F-measure from almost 5% to 143.1% for different products. The average F-measure improvement rate over all Eclipse products is 46%.

Table 4 shows that the average F-measure improvement of predicting faulty products using BRFPst+cat compared to BRFPdesc for Gnome. The results show an improvement ranging from 4% to 174.44% for different products. In average, using stack traces and categorical features improves the accuracy by 41% across all products in Gnome.

Finding 1:
The use of stack traces and categorical features (system version, severity, and platform) provides better accuracy in predicting BR product fields compared to the use of BR descriptions.

RQ2. Can stack traces and categorical features be used to predict the <u>component</u> field of a BR and if so, what would be the accuracy and how does it compare to the use of BR descriptions?

Table 3 shows the macro average F-measure improvement to predict bugs components using BRFPst+cat compared to

BRFPdesc for Eclipse. Based on the results, the improvement ranges from 0% to 42%. Moreover, the macro average F-measure of the components of each product has improved using the proposed approach. Table 5 presents the results for the Gnome dataset. The improvement ranges from 14.73% to 50%. For the components of each product, the proposed approach outperforms BRFPdesc.

Finding 2:
The use of stack traces and categorical features (system version, severity, and platform) provides better accuracy in predicting BR component fields compared to the use of BR descriptions.

RQ3. How does our approach compare to a random classifier?

Table 2 and Table 4 show the product prediction accuracy of our approach using stack trace and categorical features compared to a random approach for Eclipse and Gnome respectively. Our approach outperforms a random approach when predicting faulty products in Eclipse and Gnome datasets with an average improvement of 200% and 250% respectively. Similarly, as we can see in Table 3 and Table 5, our approach outperforms a random classifier for predicting fault components by 205% and 391% for Eclipse and Gnome respectively.

Finding 3:
Our classification approach combined with stack traces outperforms significantly a random classification method.

4.2 Discussion

Despite the overall excellent performance of BRFPst+cat when applied to both Eclipse and Gnome datasets, there were cases where our approach did not perform well when predicting the right components. These cases are shown in the add-on material submitted with this paper (in the paper, we only show macro_averages). We found that this happens mainly when (1) the number of stack traces is small, and/or (2) the BR descriptions contain bug reproduction steps or source code information; these BRs are usually submitted by developers.

For the Eclipse product "E4", components "Resources" and "Tools", BRFPst+cat achieved lower accuracy than BRFPdesc. The improvement is -8.35and -13% respectively. Our model needs a sufficient number of stack traces to generalize and make an accurate prediction. When the number of stack traces is low, which is the case for these components, it is less likely to have shared functions (used as features) among stack traces due to uniqueness of stack traces compared to BR descriptions, which contain common words since they are written in a natural language. Because the number of stack traces of BRs associated with these components is low, the feature vectors built using functions in stack traces did not properly characterize the corresponding BRs in the vector space, resulting in a cosine similarity among stack traces based on Equation (3) that converged to zero.

For the "Incubator" component of the Eclipse "PDE" product, the use of BR descriptions yields better results than our approach (improvement achieved by our approach of -33.30%). By further investigating the BRs associated with this component, we observed that they contain detailed steps on how to reproduce the bug embedded in the BR description. An example of a snippet of BR for this component is shown in Table 6. The same observation holds for the "Device Kit" component of the Eclipse Equinox product where most BRs related contain the bug reproduction steps in their description (see BR #192746 in Table 7 for an example).

Table 6. Eclipse BR #213234

BR Field	Value
Product	PDE
Component	Incubators
Header	[api tooling] invalid thread access setting up API tooling
Description	steps: 1. open the editor for an element that will have a source tag added to it by the wizard 2. make a change to the type and do not save it 3. start the setup wizard specific example I used to reproduce: 1. get debug.ui from head 2. open FileLink and make a change, do not save 3. run the setup wizard on debug

For the Gnome dataset, our approach outperforms BRFPdesc in most cases except for predicting the components "gnome-games" and "gconf-editor". We found that this is due to the fact that the BRs of these components contain information which are mostly technical and different from the categorical information provided in the bug tracking system. For example, in BR #408425 (see Table 8), we have information such as distribution of the Linux environment, release information of the OS, memory status, etc. The same observation holds for the "gconf-editor" component of the Gnome "Applications" product.

4.3 Implications and Limitations

In this section, we discuss the implications of our findings.

On stack traces: Our findings clearly show the importance of stack traces in predicting the product and component fields of BRs. This confirms the need to collect stack traces whenever a bug report is submitted. Traces should not be copied and pasted in BR descriptions, as it is the case in many bug tracking systems. Bug report tracking systems should be designed in a way that facilitates the collection and mining of stack traces. It is recognized that stack traces require storage and processing capabilities because of their size. For Mozilla products, for example, stack traces are only kept for one year because of the overhead caused by managing these traces [14]. Therefore, simply collecting traces may not be sufficient, we need to investigate better ways to structure their content by reducing noise and other elements that may not be needed to characterize the corresponding BRs. We should also look at using trace abstraction and reduction techniques such as the ones presented in [31]. The authors showed that abstractions constructed from stack traces can be used in an efficient way to detect duplicate BRs.

Table 2. Product prediction accuracy for Eclipse

Product	BR Descriptions			BR Stack traces and categorical features			Random	Improvement over description	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Platform	46%	68%	54.80%	55%	68%	60%	50.2%	9.50%	19.5%
JDT	47%	68%	55.50%	69%	68%	68.40%	30.1%	23.20%	127.2%
PDE	27%	23%	24.80%	50%	76%	60.30%	9%	143.10%	570%
Equinox	34%	37%	35.40%	60%	48%	53.30%	7.9%	50.50%	574.6%
E4	55%	47%	50.60%	56%	50%	52.80%	2.7%	4.34%	1855.5%
AVERAGE	42%	49%	44%	58%	62%	60%	20%	46%	200%

Table 3. Component prediction accuracy (Eclipse)

Product	BR Descriptions			BR Stack Traces with categorical features			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Platform	35%	29%	31.71%	50%	41%	45.05%	5.8%	42.04%	676%
JDT	52%	49%	50.45%	67%	64%	65.46%	20%	29.74%	227%
PDE	61%	56%	58.39%	68%	60%	63.75%	25%	9%	155%
Equinox	55%	37%	44.23%	63%	43%	51.11%	11%	15.53%	364%
E4	73%	58%	64.64%	78%	56%	65.19%	33%	0%	97%
AVERAGE	55%	46%	50%	65%	53%	58%	18.96%	19%	205%

Table 4. Product prediction accuracy (Gnome)

Product	BR Description			Bug			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Deprecated	71%	75%	72.94%	81%	77%	78.94%	15.8%	8.23%	399.62%
Core	65%	73%	68.76%	70%	73%	71.46%	35.7%	3.93%	100.17%
Other	60%	65%	62.4%	68%	68%	68%	16.2%	8.97%	319.75%
Platform	69%	38%	49%	92%	72%	80.78%	2.3%	64.83%	3412.17%
Applications	74%	62%	67.47%	83%	72%	77.1%	29.6%	14.29%	160.47%
Infrastructure	35%	52%	41.83%	46%	47%	46.49%	0.1%	11.13%	46390.00%
Bindings	36%	18%	24%	78%	57%	65.86%	0.07%	174.44%	93985.71%
AVERAGE	59%	55%	55%	74%	67%	70%	20%	41%	250.00%

Table 5. Components prediction accuracy (Gnome)

Product	BR Descriptions			BR Stack Traces with categorical features			Random	Improvement over description model	Improvement over random
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Accuracy		
Deprecated	59%	59%	59%	96%	71%	81.63%	5%	38.35%	1532.60%
Core	65%	41%	50.28%	73%	62%	67.05%	11%	33.35%	509.55%
Other	50%	42%	45.65%	75%	63%	68.48%	5.5%	50%	1145.09%
Platform	62%	48%	54.10%	85%	67%	74.93%	5.8%	38.51%	1191.90%
Applications	58%	47%	51.92%	68%	53%	59.57%	8.3%	14.73%	617.71%
Infrastructure	60%	53%	56.28%	72%	63%	67.20%	50%	19.40%	34.40%
Bindings	61%	55%	57.84%	78%	66%	71.50%	14.2%	23.61%	403.52%
AVERAGE	59%	49%	54%	78%	64%	70%	14.25%	31%	391.23%

Table 7. Eclipse BR #192746

BR Field	Value
Product	Equinox
Component	Incubator.DeviceKit
Header	Try to create new DK project fails
Description	Steps to recreate: 1. New->Other->Device Kit->Device Kit Components->Connection 2. Connection Name = Something 3. Finish becomes enabled click to attempt to create the connection 4. Device Kit Error Found this error in the log !SESSION 2008-02-26 23:40:00.369 ---------- eclipse.buildId=M20071023-1652 java.version=1.5.0_13 java.vendor=Apple Computer, Inc. BootLoader constants: OS=macosx, ARCH=x86, WS=carbon, NL=en_US

Table 8. Gnome BR #408425

BR Field	Value
Product	Deprecated
Component	gnome-games
Header	Crash while closing the window
Description	If you click the New button in the toolbar, so that the 'new game' dialog shows up, and then you close the window using the window manager close button, HEAD crashes. Distribution: Fedora Core release 6 (Zod) Gnome Release: 2.17.90 2007-02-10 (JHBuild) BugBuddy Version: 2.17.3 System: Linux 2.6.19-1.2895.fc6 #1 SMP Wed Jan 10 19:28:18 EST 2007 i686 X Vendor: The X.Org Foundation X Vendor Release: 70101000 Selinux: Enforcing Accessibility: Enabled GTK+ Theme: Clearlooks Icon Theme: gnome

On BR categorical attributes: We showed that categorial attributes namely version, platform, and severity, enhance the prediction accuracy. The problem is that these attributes themselves may be entered incorrectly, which is a threat to validity for our approach (see next section). Our findings strengthen the need to have these attributes automatically and correctly generated. Users should never have to enter these attributes.

On BR descriptions: Although many studies showed that BR descriptions are useful in various BR triaging activities, after working with many large BR repositories (some of them from industry) we remain very skeptical as to the sole use of descriptions for prediction tasks. We found many BRs where the descriptions consist mainly of snippets of stack traces. We only found descriptions that are poorly written and hard to understand. In addition, unless there are clear guidelines on how to write proper descriptions and that these guidelines are enforced, descriptions remain largely inaccurate and imprecise.

Limitation 1 - Sparsity of stack traces: The main limitations of our approach is that, for the time being, only a small portion of BRs come **with** stack traces. Our Eclipse dataset contains only 10% of BRs with stack traces. We hope that the findings of this paper will encourage the collection of stack traces in a systematic manner.

Limitation 2 - Scalability: In our approach, we compare each incoming BR with all the BRs that were submitted before. This process updates the feature vectors by adding the new functions that were not learned from the previous BRs and by updating the TF-IDF weights. We opted for this process to simulate how BRs are processed in real world. This method, however, incurs a high computational cost.

5 Threats to Validity

Our proposed approach and the conducted experiments are subject to threats to validity.

There is a threat to validity with respect to the training dataset we used to optimize the free parameters w1, w2, w3, and w4 used to weigh the features in the linear combination of stack traces and categorical attributes (see Equation (4)). A different threshold may lead to different results.

Another threat to internal validity exists in the way we implemented the approach for extracting words from BR descriptions. We simply tokenize each BR description and used the extracted words to form feature vectors. We did not resort to any natural language processing method. The use of a powerful natural language processing method may result in better performance of an approach that uses BR descriptions.

Finally, the misclassification cost in Equation (10) is calculated using our own proposed heuristic. However, this parameter could be adjusted using an exhaustive domain search or training machine learning methods. Using a more optimal parameter could further enhance the product and component prediction capability of our approach.

While we managed to work on two large bug repositories, which are extensively used in the literature, the low number of stack traces is a threat to validity. We cannot claim that these results apply to other bug reports. Therefore, the presented observations and findings are best interpretable with respect to the particular bug reports we chose and studied.

6 Related Work

Bettenburg et al. [2] studied the quality of BRs and showed that there is an important gap between what users provide as an input and what developers need to fix a bug. They argued that since users do not usually have technical knowledge about the system, it is very difficult for them to properly report the faulty products and components.

Guo et al. [6] introduced the bug pong concept as sending the bug between development teams similar to ping pong ball. The authors

showed that it happens when a faulty component is not correctly identified in the BR. They also showed that incorrect selection of the faulty components increases the bug processing time.

Breu et al. [3] showed that the questions asked when submitting BRs can be grouped into eight categories: missing information, clarification, triaging, debugging, correction, status inquiry, resolution, and administration. They showed that triaging questions are mostly due to the fact that users enter the wrong products and components when reporting a bug.

Somasundaram et al. [17] showed that the component field in a BR helps triagers route the bug to the right development team. They have also shown that incorrect component categorization often delays the resolution of the BR. They used the description of bug reports and applied Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) with SVM and LDA with Kullback divergence (KL). They observed that LDA-KL produces more stable results than SVM.

Shihab et al. [15] showed that the processing time of re-opened bug reports is twice more than the processing time of regular bugs. They observed that the description, fixing time, and the component of BRs are the most important factors in determining whether a bug will be re-opened. They showed that the reporters' name is not an important factor in predicting re-opened bugs. They have structured their study in four dimensions: work habit (weekday of closing bugs), bug reports faulty component, bug fix (time spent to fix the bug), and people (experience of the bug fixer). They extracted features from these dimensions to build up a decision tree to predict whether a bug will be re-opened. They showed that both the BR dimension and faulty components have the best F-measure in predicting re-opened bug reports.

Giger et al. [5] showed that, in Firefox BRs, the faulty component field is the most important factor in determining how fast a bug will be fixed. They showed that, in Gnome BRs, the component is the second most important field in determining the fixing time of a bug.

Sureka [12] showed that the most important feature to localize the fault of a bug is the component. He showed that the component field is usually assigned wrongly by most users. He has revealed that the highest frequency of reassignment after the assignee field is the component field. The author used a statistical and a probabilistic model applied to the title and description of BRs to predict faulty components. The study achieved 42% accuracy when predicting the component field of a BR. The author also showed that TF-IDF applied to the description of BRs performs the same as Dynamic Language Model (DLM). To detect whether the component field of a BR will be reassigned, his approach has achieved 42% of accuracy.

Lamkanfi et al. [7] revealed that the component field in Eclipse and Mozilla BRs tends to be regularly reassigned. They extracted all initial values of BR information including component, reporter, operating system, version, severity, and BR summary to decide whether the component of a bug will be reassigned. Their classifier has predicted reassigned bugs with an F-measure of over 44% and not reassigned bugs with an F-measure of over 83%.

Xia et al. [27] showed that 80% of bugs have their fields reassigned. They have also revealed that the bugs which are reassigned take longer time to be fixed. They showed that the product and component fields usually get reassigned together.

Xia et al. [20] used a multi label learning algorithm (ML.KNN) to predict reassignment of BR fields. To overcome the unbalanced dataset problem, they have used the IM-ML.KNN classification approach. Their approach achieved an F-measure ranging from 56% to 62%.

Wang et al. [19] used BR descriptions and summaries to predict the component field of a BR. They have created feature vectors using words in the descriptions and summaries. They weighed the feature vectors using TF-IDF. They compared the performance of support vector machines and Naïve Bayes machine learning techniques. They applied their approach to Eclipse BRs and showed that SVM models outperform Naïve Bayes models in predicting components.

Maiga et al. [32] showed that severity fields of the bug reports are not usually precisely chosen by the software users in industry.

Sabor et al. [33] showed that product and component field of bug reports could be predicted more accurately using deep learning methods.

7 Conclusion

In this paper, we proposed an approach to predict product and component fields of BRs that leverages stack traces and categorical features instead of BR descriptions. In our approach, we have used a linear combination of stack traces and categorical features similarity to predict products and components. We tested our approach to two Eclipse and Gnome BR repositories. Oure experiment showed that our approach could predict the faulty product and component by an average F-measure of 65%. We showed that our approach outperforms the one which uses bug report descriptions to predict faulty product and components of bugs by 35% in average. Our approach could be effectively used to predict faulty product and components of a bug to eliminate the overhead of wrong assignment of the bugs to the development teams. As a future work, we also plan to use a training model to obtain the optimized misclassification cost for each product and component. Furthermore, we plan to use more advanced machine learning techniques such as deep learning.

ACKNOWLEDGMENTS

This research has been partly supported by the Natural Science and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change request," in Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON), 2008.

[2] N. Bettenburg, S. Just, A. Schr¨oter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?” in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. SIGSOFT ’08/FSE-16. New York 2008, pp. 308–318.

[3] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, “Frequently asked questions in bug reports,” University of Calgary, Technical Report, 2009.

[4] J. Gou, L. Du, Y. Zhang, T. Xiong, « A New Distance-weighed K-nearest Neighbor Classifier,” Journal of Information & Computational Science, 2012, pp. 1429-1436.

[5] E. Giger, M. Pinzger, and H. Gall, “Predicting the fix timeof bugs,” in Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, 2010, pp. 52–56.

[6] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ““not my bug!” and other reasons for software bug report reassignments,” in Proceedings of the Conference on Computer Supported Cooperative Work, 2011, pp. 395–404.

[7] A. Lamkanfi and S. Demeyer, “Predicting reassignments of bug reports an exploratory investigation,” in Proceedings. of the 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 327–330.

[8] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, 2010, pp. 1–10.

[9] A. Lamkan, S. Demeyer, Q. D. Soetens, and T. Verdonck, “Comparing mining algorithms for predicting the severity of a reported bug,” in Proc, of the European Conference on Software Maintenance and Reengineering , 2011, pp. 249-258

[10] C. D. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval. Cambridge University Press, NY, USA, 2008.

[11] F. Provost, T. Fawcett. Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking, O'Reilly Media, 2013.

[12] A. Sureka, “Learning to classify bug reports into components,” in Proceedings on the 50th International Conference on Objects, Models, Components, Patterns: 50th International Conference (TOOLS), 2012, pp. 288–303.

[13] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces,” in Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, 2016, pp. 96–105.

[14] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, “Automatically analyzing groups of crashes for finding correlations,” In Proceedings 2017 11th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2017, pp. 717–726, 2017.

[15] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, “Predicting re opened bugs: A case study on the Eclipse project,” in Proceedings of the 17th Working Conference on Reverse Engineering (WCRE), 2010, pp. 249–258.

[16] C. Sun, D. Lo, S. C. Khoo and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in Proceedings. of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 253-262.

[17] K. Somasundaram and G. C. Murphy, “Automatic Categorization of Bug Reports Using Latent Dirichlet Allocation”, in Proc. of the Proceedings of the 5th India Software Engineering Conference (ISEC), 2012, pp. 125-130.

[18] Y. Tian, D. Lo, and C. Sun, “Information retrieval based nearest neighbor classification for fine-grained bug severity prediction,” in Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), 2012, pp. 215–224.

[19] D. Wang, H. Zhang, R. Liu, M. Lin, and W. Wu, "Predicting Bugs' Components via Mining Bug Reports," Journal of Software, 7(5), 2012, pp. 1149-1154.

[20] X. Xia, D. Lo, E. Shihab, and X. Wang, “Automated bug report field reassignment and refinement prediction,” In IEEE Transactions on Reliability, 65(3), 2016, pp. 1094–1113.

[21] J. Xie, M. Zhou and A. Mockus, "Impact of Triage: A Study of Mozilla and Gnome," in Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 247-250.

[22] G. Yang, T. Zhang, and B. Lee, “Towards semi-automatic bug triage and severity prediction based on topic model and multi feature of bug reports,” in Proceedings of the 38th Annual Computer Software and Applications Conference , 2014, pp. 97–106

[23] Z. Qin, A. T. Wang, C. Zhang, and S. Zhang. Cost-Sensitive Classification with k Nearest Neighbors, In Proceedings of 6th International Conference on Knowledge Science, Engineering and Management , pages 112–131, , 2013.

[24] J. Gou, L. Du, Y. Zhang, T. Xiong, « A New Distance-weighed K-nearest Neighbor Classifier,” Journal of Information & Computational Science, 2012, pp. 1429-1436

[25] T. Zhang, G. Yang, B. Lee, and A. T. S. Chan, “Predicting severity of bug report by mining bug repository with concept profile,” in Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC), 2015, pp. 1553–1558

[26] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in Proceedings of European Conference on Software Maintenance and Reengineering, 2013, pp. 69-78.

[27] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, “An empirical study of bug report field reassignment,” in Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering ,2014, pp. 174–183.

[28] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, “CrashLocator: locating crashing faults based on crash stacks,” *Proc. 2014 Int. Symp. Softw. Test. Anal. - ISSTA 2014*, pp. 204–214, 2014.

[29] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, “A bug reproduction approach based on directed model checking and crash traces,” *J. Softw. Evol. Process*, vol. 29, no. 3, p. e1789, 2017.

[30] N. Chen and S. Kim, “Star: Stack trace based automatic crash reproduction via symbolic execution,” *PhD Thesis, Honk Kong Univ. Sci. Technol.*, vol. 41, no. 2, pp. 198–220, 2015.

[31] K. Koochekian Sabor, A. Hamou-lhadj, and A. Larsson, “DURFEX : A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports,” in Proceedings of CASCON 2016, pp. 240–250, 2017.

[32] A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K. Koochekian Sabor and A. Larsson, "An empirical study on the handling of crash reports in a large software company: An experience report," *2015 IEEE International Conference on Software Maintenance and Evolution*, Bremen, 2015, pp. 342-351.

[33] K. K. Sabor, M. Nayrolles, A. Trabelsi and A. Hamou-Lhadj, "An Approach for Predicting Bug Report Fields Using a Neural Network Learning Model," *2018 IEEE International Symposium on Software Reliability Engineering Workshops*, Memphis, TN, 2018, pp. 232-236.