Automatic Prediction of the Severity of Bugs Using Stack Traces

Korosh Koochekian Sabor
Software Behaviour Analysis (SBA) Lab
ECE, Concordia University
Montréal, Québec, Canada
k_kooche@ece.concordia.ca

Mohammad Hamdaqa
Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario, Canada
mhamdaqa@uwaterloo.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Lab
ECE, Concordia University
Montréal, Québec, Canada
abdelw@ece.concordia.ca

ABSTRACT

The severity of a bug is a measure of how a defect affects the functionality of a system. Developers refer to the severity of the reported bugs to prioritize the handling of bug reports. The process of assigning a severity level to a bug is performed manually, often by inexperienced users, making it time consuming and error prone. Existing techniques for automatically predicting the severity of bugs rely on text mining and information retrieval algorithms applied to the description of the bugs. The problem is that the description tends to be too informal and not quite reliable. In this paper, we show how information found in stack traces (a more formal source of data containing the history of function calls to the function in which the crash happened) can be used to automatically predict the severity of bugs. Our experiments with Eclipse bug reports submitted between 2001 to 2015 show that stack traces are a better feature for predicting the severity of bugs than the bug description.

Keywords

Bug Severity, Mining Software Repositories, Software Maintenance, Machine Learning

1. INTRODUCTION

When a system crashes, the system users have the option to report the bug that caused the crash using a bug tracking system that usually comes installed with the system. Example of bug tracking systems include the Windows Error Reporting tool[1], the Mozilla crash reporting system[2], and the Ubuntu's Apport crash reporting tool[3]. Bug tracking systems are designed to capture important information about the crash such as the description of the bug, the stack trace, a summary of the bug, and the severity of the bug. Developers use this information to resolve the underlying faults.

Despite the existence of bug tracking systems, bug handling is considered a tedious and expensive task [2, 5]. For example, the Eclipse platform receives nearly 100 reports each day [3]. Handling all the bug reports manually and assigning them to expert developers requires an excessive amount of effort.

To balance the workload of the developers and optimize the bug handling process, not all the bugs are handled at the same time. Bugs need to be prioritized. One way to prioritize bugs is by assigning to them a severity level. The bug severity is assigned to a bug by the user who submits the bug report. Bug severity is a measure of the impact of a bug on the execution of the system [7].

It can be used as an indicator of how soon a bug needs to be fixed [20]. Bugs with higher severity levels are those that affect the system in a serious way and, therefore, should be handled earlier than those with a low severity level. In the Eclipse bug repository (used as a dataset to evaluate our approach), a bug severity can take one of the following labels: Blocker, Critical, Major, Normal, Minor, Trivial or Enhancement [7]. Bugs with the Blocker severity level have the highest importance, whereas those with the Enhancement severity level should be given the lowest priority.

In theory, a user assigns the severity of a bug according to its defective appearance. In practice, however, this does not seem to be the case. Many users simply select the default severity label [8]. For example, in Eclipse, when submitting a bug, the 'Normal' severity label is the default choice. It is usually selected arbitrarily by the users [18]. As a result, the severity that is submitted in the bug report does not reflect the real impact of the bug on the system, which affects the bug handling process. To address this issue, several severity prediction techniques that aim to automate the process of assigning severity labels to bug reports have been proposed (e.g., [1, 8, 12]). The common practice is to classify an incoming bug based on historical data. These techniques use the bug description as a feature. Bug descriptions, however, are informal and contain a lot of noise.

There are two types of severity prediction techniques based on the level of granularity of the severity labels (i.e., coarse grained and fine grained severity prediction). In the literature, many studies focused on coarse-grained severity prediction, in which, the severity of a bug takes two values: non-severe or severe. In these studies, severe bugs are bugs with Blocker, Critical or Major severity, while non-severe bugs are bugs with Minor or Trivial severity [12]. Non-severe bugs can stay in a bug routing queue for a longer period of time, while severe bugs must be routed immediately to the developers to provide fixes. In fine-grained severity prediction, each of the severity labels is considered separately and is not abstracted out to non-severe and severe

[1]http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx

[2]http://crash-stats.mozilla.com

[3]https://wiki.ubuntu.com/Apport

categories. This helps the bug triagers optimize resources more efficiently.

In this paper, we propose a new fine-grained bug severity prediction technique based on stack traces of the bug reports. Our approach maps stack traces into term vectors weighed by term frequency–inverse document frequency (tf.idf). The term vectors are used to build a training model, which is used later to predict the severity of incoming bug reports. The mapping of stack traces into term vectors is discussed in more detail in Section 2. We show that stack traces when used as features for a classification technique perform better that bug descriptions.

To the best of our knowledge, this is the first time that stack traces are used for predicting the severity of bug reports. We show that using stack traces can improve the prediction of the bug severity and reduces the prediction error rate caused by the informality and inaccuracy of the bug report descriptions. The proposed approach uses K-nearest neighbors to predict the severity of the incoming bug reports. We evaluate the effectiveness of our approach by applying it to the Eclipse bug repository that contains 455,700 bug reports. We show that using stack traces to predict bug severities results in a higher precision and recall compared to using bug descriptions.

The paper is organized as follows. In Section 2, we introduce preliminary concepts. After that, our severity prediction approach is explained in Section 3. We elaborate on the dataset and the evaluation results in Section 4. Section 5 highlights the related work. Finally, we summarize this paper and printout future works in Section 6.

2. PRELIMINARIES

This section explains how to map a stack trace into a term vector weighed by term frequency–inverse document frequency *(tf.idf)*.

Let $\mathrm{T} = \mathrm{f}_{1,}\mathrm{f}_{2,} \ldots\ldots, \mathrm{f}_{\mathrm{L}}$ be a stack trace of function calls f_1 to f_L. T is of length L. The stack trace T is generated by a bug in a system from an alphabet Σ of size m = $|\Sigma|$ that represents unique function names in the system. The collection of K traces that are generated by the process (or system) of interest and then provided for designing the severity prediction system is denoted by $\Gamma = \mathrm{T}_{1,}\mathrm{T}_{2,} \ldots\ldots, \mathrm{T}_\mathrm{K}$.

The term vector maps each trace $T \in \Gamma$ into a vector of size m functions, $T \rightarrow \emptyset(T)_{fi\in\Sigma}$, where each function name $f_i \in \Sigma$ in the vector is assigned a binary flag depending on its appearance (one) or not (zero) in the trace T. The term vector can be weighted by the term frequency (*tf*):

$$\phi_{tf}(f, T) = freq(f_i); i = 1, \ldots, m \qquad (1)$$

where *freq(f$_i$)* is the number of times the function f_i appears in T normalized by L (the total number of function calls in T).

The term frequency considers all terms as equally important across all documents or collection of traces (Γ). However, rare terms that appear frequently in a small number of documents convey more information than those that are frequent in most documents. The inverse document frequency (*idf*) is proposed to increase (or decrease) the weights of terms that are rare (or common) across all documents. The term vector weighted by the *tf.idf* is therefore given by:

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

where the document frequency $df(f_i)$ is the number of traces T_k in the collection of Γ of size K that contains function name f_i. A high weight in *tf.idf* is thereby given to function names that are frequent in a particular trace $T \in \Gamma$, but appear in few or no other traces of the collection Γ.

3. PROPOSED APPROACH

In this paper, we use an online severity prediction approach to predict the severity of each incoming bug report to the bug tracking system. Figure 1 shows an overview of the proposed approach. The approach consists of two phases: a training phase and a testing phase.

In the training phase, a training model is constructed by first extracting all the stack traces. Then, for each trace $\mathrm{T}_{\mathrm{i},} \in \Gamma$, a feature vector is constructed and then assigned weights according to the term vector weighing strategy explained in Section 2. The output of the first phase will be an adjacency matrix, in which each trace is represented as a term frequency vector of features (function calls).

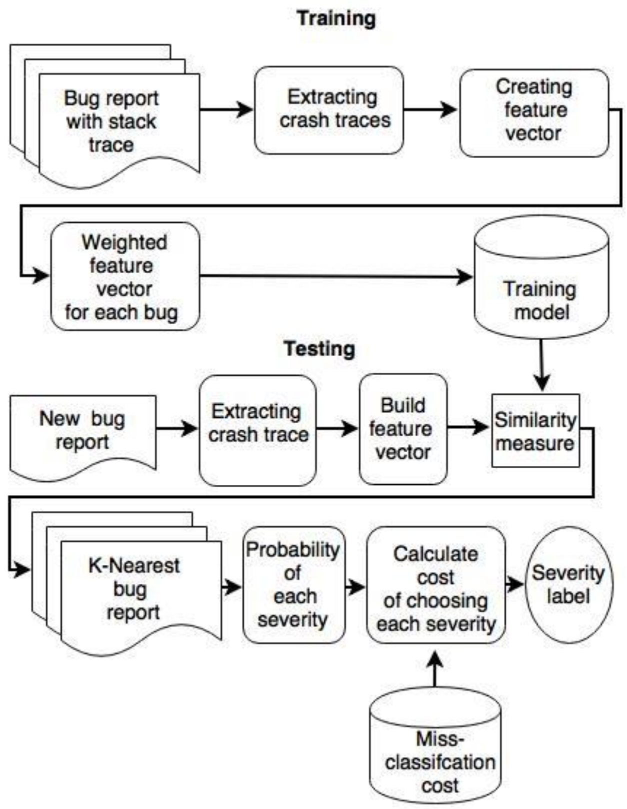

Figure 1. Proposed approach

In the testing phase, with the arrival of each new bug report, B_i, a feature vector of the corresponding bug report is constructed. Similarly, each feature in the feature vector is weighed according to the frequency and inverse document frequency of that feature as explained earlier. Then, the constructed feature vector of the new bug report B_i is compared to the feature vector of each previous bug reports in the training dataset (i.e., the traces adjacency matrix).

The distance between two bug reports is measured as the distance between their corresponding vectors containing features weighed using their frequency of occurrence and inverse document frequency. The distance between the two vectors is calculated using the cosine similarity as shown in equation (3).

Given $V_1 =< w_{11}, w_{12}, \ldots.. w_{1n} >$ and $V_2 =< w_{21}, w_{22}, \ldots.. w_{2n}>$ the cosine similarity is calculated as in Equation (3) [10,11]:

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

The cosine similarity between two vectors is the cosine of the angle between two vectors. It is equal to the dot product of the two vectors divided by the multiplication of their sizes.

Once the distance between the victors is calculated, the K-Nearest Neighbor (KNN) algorithm is used to return the list of the most similar bug reports. KNN is an instance-based lazy learning algorithm [16] that is usually used in classification and regression. The algorithm returns the (k) most relevant instances for a given feature vector. As a classifier, the KNN algorithm consists of two phases. In the first phase, a distance measure or similarity metric is used to calculate the distance between a specific instance and the training data instances (e.g., cosine similarity). Accordingly, based on the value of (k), which is a constant value that defines the number of neighbors that need to be returned, the algorithm returns the nearest relevant instances. In the second phase, a voting technique (e.g., majority voting) uses the labeled returned instances to classify (X), where X is the instance in question (i.e., the instance that corresponds to B_i) by setting a label to it. The label of instance X can be determined given the labels set C by majority voting as in (4) [14].

$$C(\mathrm{X}) = \underset{c_j \in C}{argmax} \quad score\ (c_j, neighbors_k(X)) \quad (4)$$

In (4), $neighbors_k(X)$ is the K nearest neighbors of instance X, *argmax* returns the label that maximizes the score function, which is defined in (5) [14].

$$\text{Score}\ (c_j, N) = \sum_{Y \in N} [class(y) = c_j] \ (5)$$

In (5), [class(y) = c_j] is considered to be 1 if class(y) = c_j and 0 otherwise. According to (5), the score for each label is calculated based on frequency of the appearance of that label in the retrieved list. The label with highest frequency is chosen as the label of the incoming bug report.

According to (4) and (5), the K-nearest neighbor ignores the distance of the neighbors in the list and choses the label of the instance based on the frequency of appearance of each label. In our approach, when the K-nearest neighbor of an incoming bug is retrieved, we need to give more weight to the severity label of the bugs that are closer to the incoming bug. The rational is that a bug in the training set, which is closer to the bug in the test set (X) is more probably to be caused by the same underlying fault. It should have the same severity as the bug in the test set.

If we assume the distance of the nearest bug in the sorted list as $dist_1$ and distance of the farthest bug in the retrieved list as $dist_k$ then the weight of each bug severity label in the list can be calculated by (6) [6], where $dist_i$ is the distance of bug *i*.

$$\boldsymbol{w_i} = \begin{cases} \frac{\boldsymbol{dist_k - dist_i}}{\boldsymbol{dist_k - dist_1}}, & \boldsymbol{if\ dist_k \neq dist_1} \\ \boldsymbol{1}, & \boldsymbol{if\ dist_k = dist_1} \end{cases} \quad (6)$$

According to (6), the weight of each instance (i) in the retrieved list is equal to its distance ($dist_i$) subtracted from the distance of the farthest instance ($dist_k$) normalized by the difference of the value of nearest and farthest distance ($dist_k$) *and* ($dist_1$) respectively. According to (6), instances that are closer to the incoming bug report will have higher weights. Based on (6), the score function in (5) can be updated as shown here in (7) [14]:

$$\text{Score}\ (c_j, N) = \sum_{Y \in N} w(x, y) \times [class(y) = c_j] \quad (7)$$

In (7), *w(x,y)* is the weight of each instance *y* in the retrieved list, which is calculated by (6) according to its distance to the test instance *x*. After calculating the weight of each label, the label with the highest weight is chosen as the label of the instance in the test set according to (4).

In a normal situation, the distribution of severity labels can be unbalanced (i.e., some severity labels may appear more frequently than others). For example, in the dataset used in this paper, the *Major* class label is represented by a large number of bug report instances (e.g., 6512 classes out of 11825). This is more than the total number of instances representing all the other class labels combined (e.g., 5313 classes).

The unbalanced distribution of severity labels can mislead the classification results. This is because, classifiers - when trained - usually try to increase the overall accuracy of classification, which may lead to ignoring the minority sample in the dataset. Methods such as resizing the training set, adjusting misclassification cost and recognition based learning can be used to overcome the unbalanced data distribution problem [21].

In our approach, we use cost-sensitive learning[15] to overcome the effect of an unbalanced dataset. This particular approach has been selected based on experimentation. We noticed that cost-sensitive learning performs better than dataset resizing when applied to the dataset used in this paper.

Any classifier that provides probability estimate for class labels can be transformed to a direct cost sensitive classifier using a cost matrix. Using a cost matrix, the probability of belonging to each label is replaced by the average cost of choosing that class label. A direct cost sensitive classifier does not manipulate the internal procedure of the classifier. Instead, it makes an optimal cost-sensitive prediction according to the output of the classifier.

According to Equations (4) and (7), the outcome of our classifier is the weight of each label for the test instance. To use a cost-sensitive classifier, we must transform the output of our classifier to represent the probabilities of how a test instance belongs to each of the existing classes. Consider a bug in the test dataset as B, having M classes, the classifier must provide a list of probabilities $p_1 \ldots \ldots \ldots p_m$, in which each p_i shows the probability of the bug in the test set that belongs to the ith class. Since, the summation of all probabilities should equal one (i.e., $p_1 + p_2 + \cdots + p_m$ =1), the weights need to be normalized.

The score function (7) returns the weight of the class labels for each incoming bug report. Consider $w_1 \ldots \ldots \ldots w_m$ are the weights, which are returned as the classification result. To calculate the probability of each class we use (8), where W=$w_1 + \cdots + w_m$

$$p_i = \frac{w_i}{W} \quad (8)$$

In addition to transforming the output of the classifier to probabilities, we must define a cost matrix, which contains the misclassification cost of each class label. We set the misclassification cost in a cost matrix that corresponds to the confusion matrix in Figure 2.

	Actual	
	Positive	Negative
Predicted	True positive	False positive
	False negative	True negative

Figure 2. Confusion matrix

Since correct classification is encouraged in our case, we set the cost of true positive and true negative to zero in the cost matrix. Meanwhile, the cost of the false positive and false negative will be chosen according to the classification case. This rational extends to the multiclass classification problems; in which, we set the cost of all diagonal cells in cost matrix corresponding to the confusion matrix to zero, because they represent correct classifications. Then we decide about the cost of the misclassification of the other classes according to the problem domain.

We must assign high misclassification cost to under-sampled classes and lower misclassification cost to majority classes to encourage the classifier to choose minority classes. Thus, in this paper, we chose the cost of misclassification of each class label to be reciprocal to the number of existing instances of that class divided by the number of instance of the majority class. Consider that we have *C* classes, s_j is the number of instances of class *j* in the training set and s is the number of instances of the majority class in the training set then the misclassification cost of each class C_j is calculated by (9).

$$MC_j = \frac{s}{s_j} \quad (9)$$

In the final stage of classification, the cost of each class label must be calculated and the optimal class label (class label with the lowest cost) must be chosen as the predicted label of the instance in the test set.

Consider we have M classes and the incoming bug belongs to each of these classes with probabilities of $P_1 \ldots\ldots P_m$, let 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 (10).

$$CCO_i = \sum_{j \in m \text{ and } j \neq i} CO_j \times P_j \qquad (10)$$

In (10), CCO_i is the cost of classifying the instance in the test set with class *i*. According to (10) the cost of classifying the instance in the test set to class *i* is equal to sum of multiplication of the probability by misclassification cost of all other classes. After calculating CCO for each of the classes, the class label with least CCO is chosen as the label of the test instance.

Although cost-sensitive classification could solve the problem of an unbalanced dataset, assigning improper misclassification cost to a class may deteriorate the classification accuracy considerably. To avoid misclassification that is due to the very high cost that may be associated to some class labels, we choose 10 to be the maximum misclassification cost, which can be associated to a class label.

The high level pseudo code of our approach is presented in Algorithm 1.

4. EVALUATION

In this section, we assess the accuracy of predicting the severity of a bug report using stack traces compared to using the description of bug reports. We start by presenting the dataset used in this study. Next, we explain the experimental setup. Then, we evaluate the result of our approach, followed by threats to validity. This case study addresses the following research questions:

RQ1. Can stack traces be used to predict the severity of a bug report?

RQ2. What is the accuracy of predicting the severity of a bug report using stack traces compared to using the description of bug reports?

```
Procedure
Inputs:
T as the list of stack traces of bugs in the training set.
C as the list of severity types (class labels) available in our system.
B as incoming bug report
Output:
Severity label of the incoming bug report B
Method:
1.  For each $t_i \in T$
  1.1  Calculate the distance between $t_i$ and B using (3) add them to Res
2.  Choose K items from Res with the smallest distance
3.  Weigh each of the K items according to (6)
4.  Calculate the weight score of each Label by (7)
5.  Calculate the probability of each severity label by (8)
6.  Calculate misclassification cost of each severity label by (9) and construct the cost matrix accordingly
7.  For each $c_i \in C$
  7.1  Calculate classification cost of each class $c_i$ by (10)
8.  Choose the class with the lowest classification cost
```

Algorithm 1. Pseudo code of the proposed approach

4.1 Dataset

The dataset used in this paper consists of the bug reports of the Eclipse bug repository. The dataset contains Eclipse bug reports from October 2001 to February 2015, having a total of 455,700 bugs. Among these bugs, we have 297,151 bugs labeled as Normal severity and 66,873 bugs labeled as Enhancements. We removed bugs having Normal and Enhancement severity from the dataset. The reason is that the Normal severity label comes by default. It does not necessarily reflect the severity of the bug. In other words, this label cannot (and should not) be considered as a valid severity. The same decision was made by other researchers (see [7, 8, 20]). The bug reports that come as Enhancements are not considered as bugs but opportunities for enhancing the system. The remaining dataset contains 158,549 bugs with severities other than Normal or Enhancement.

In the Eclipse bug repository, stack traces are embedded in the bug report descriptions. We used the same regular expression presented by Lerch et al. [9] to extract the content of the stack traces from the bug descriptions (see Figure 3). This regular expression can extract structural information with an accuracy of 98.5% [9].

```
[EXCEPTION] ([:] [MESSAGE])? ( [at] [METHOD] [(]
[SOURCE] [)] )+ ( [Caused by:] [TEMPLATE] )?
```

Figure 3. Regular expression used for extracting stack traces from bug report description in the Eclipse bug repository

After applying the regular expression, we had 11,925 bug reports having at least one stack trace in their description. The extracted stack traces are preprocessed to remove noise from the data. For example, we removed native methods, used to call to the Java library. Stack traces with less than three functions are also removed, since they are partial stack traces and may mislead the approach. Furthermore, recursive function calls are removed. The resulting dataset contains a total of 11,825 bug reports having a total of 17,695 stack traces in their descriptions.

The distribution of the severity labels among bugs with stack traces in our dataset is shown in Figure 4.

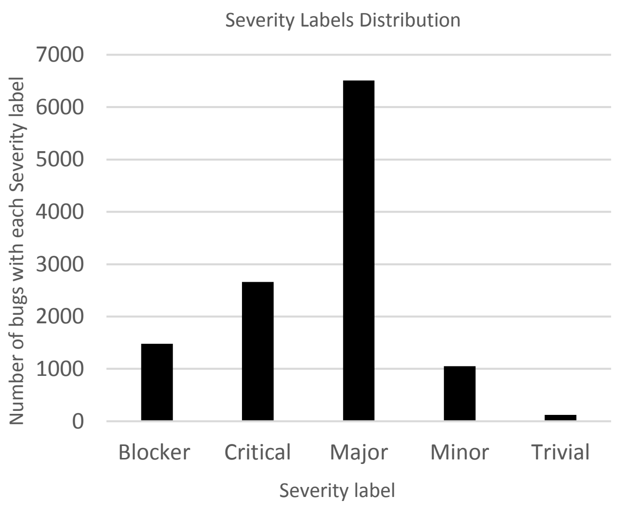

Figure 4. Distribution of severity labels in the Eclipse dataset

According to Figure 4, out of 11,825 bugs in our dataset, we have 1,479 bugs labeled as Blocker (12.5%), 2,660 bugs labeled as Critical (22.5%), 6,512 bugs labeled as Major (55%), 1,054 bugs labeled as Minor (9%), and 120 bugs labeled as Trivial (1%). The figure clearly shows that the distribution of the labels is unbalanced, favoring the Major severity label. We need to calculate the misclassification cost of each label in order to apply the cost sensitive classification algorithm shown in Algorithm 1.

Table 1. Cost matrix for the Eclipse dataset

		Actual Severity				
		Blocker	Critical	Major	Minor	Trivial
Predicted Severity	Blocker	0	2.5	1	6.2	10
	Critical	4.3	0	1	6.2	10
	Major	4.3	2.5	0	6.2	10
	Minor	4.3	2.5	1	0	10
	Trivial	4.3	2.5	1	6.2	0

The misclassification cost is calculated using Equation 9. The result is shown in the cost matrix of Table 1. The cost matrix will be used to predict the severity of an incoming bug by assigning the costs to each severity label as shown according to Table 1.

4.2 Implementation and Experimental Setup

To evaluate our approach, we implemented the algorithm depicted in Figure 1. Then, we used the following three different scenarios for predicting the severity of the incoming bug report:

(1) Index stack trace and query stack trace: In this scenario, we first index the function names of the stack traces in the description of the bug reports in the training set using the feature vector weighing technique explained earlier. Then, we query the function names of the stack trace of the incoming bug reports.

(2) Index description and query description: In this scenario, we index the description of the bug reports (including the stack trace, since stack traces in Eclipse are embedded in the description) in the training set, then for each incoming bug report, we query its description. It is noted that when using the description, each word is considered a feature.

(3) Index description and query stack trace: In this scenario, we index the description of the bug reports (including the stack trace) in the training set, then query the function names of the stack trace of the incoming bug reports. When indexing the description of the bug reports in the training set, each word is a feature. On the other hand, when querying the stack trace of the incoming bug report, each function name is considered as a feature.

In all scenarios, we used TF-IDF (Equation (2)) to weigh the feature vector and cosine similarity (Equation (3)) to return K-nearest neighbor of the new incoming bug report.

4.3 Evaluation Metrics

To compare the different scenarios, we used precision, recall and F-measure metrics. These metrics are widely used in the literature [7, 17, 20] to evaluate the ability of a classifier to predict the severity of all of the bugs correctly, and minimize the number of incorrectly-predicted severities.

Precision is defined as the ratio of the number of bugs for which we correctly predict the severity label to be S_L to the total number of bugs for which we predict that they should have a severity label S_L. We compare the precision for each severity label separately.

$$Precision\ (S_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ S_L}{\#\ of\ bugs\ predicted\ to\ have\ label\ S_L} \qquad (11)$$

Recall is defined as the ratio of the number of bugs for which we correctly predict the severity label to be S_L to the number of bugs that actually have the severity label S_L (ground truth). Similar to precision, we calculate the recall for each severity label separately.

$$Recall\ (S_L) = \frac{\#\ of\ bugs\ correctly\ predicted\ with\ label\ S_L}{\#\ of\ bugs\ actually\ having\ label\ S_L} \qquad (12)$$

We build a confusion matrix s for each severity label separately and calculate the precision and recall from each confusion matrix. Precision and recall values provide different perspectives about the prediction results. While precision measures correctness, recall measures the completeness of the results. To have a better perception of the quality of the prediction results, the values of both precision and recall should be combined. One way to combine both of them is through the F-measure, which is the harmonic mean of precision and recall [4]. F-Measure is calculated according to Equation (13).

$$F - measure\ (S_L) = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (13)$$

The F-measure decreases if the values of the precision and recall decrease and vice versa.

4.4 Evaluation Results

Perhaps the most important factor in the K-nearest neighbor algorithm is the size of the returned list of neighbors (K). Thus, after each of the scenarios explained earlier, we calculate precision, recall and F-measure for different values of (K). Figures 5 to 9 show the F-measure values for each severity label when we vary the value of (K) from 1 to 10.

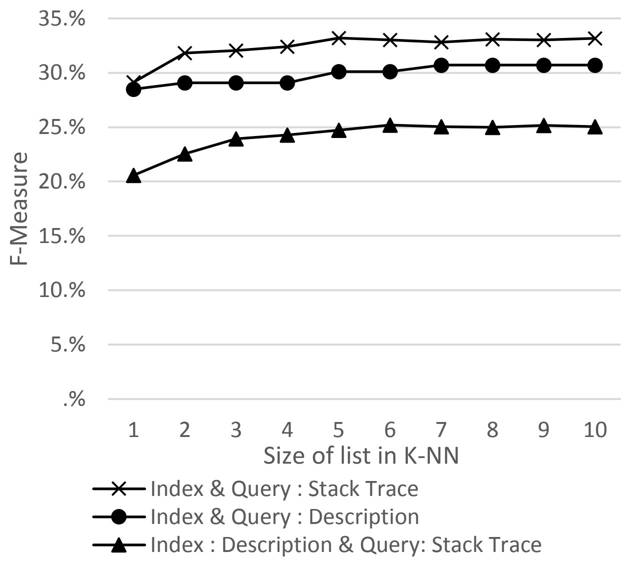

Figure 5. F-Measure of predicting Critical severity with varying list size

As shown in Figure 5, for Critical severity, the prediction results were best for the first scenario (i.e., when indexing and querying is done based on stack traces using function names as features). Furthermore, the prediction results for the second scenario (i.e., indexing the description of bug reports in the training set and querying description of the new bug report) outperforms those of the third scenario (i.e., indexes descriptions and queries the stack trace). These results confirm the fact that the content of bug reports description (without the stack traces) does not contribute to the prediction results. In fact, this extra information may mislead the severity prediction algorithm and deteriorates the performance of the severity prediction method.

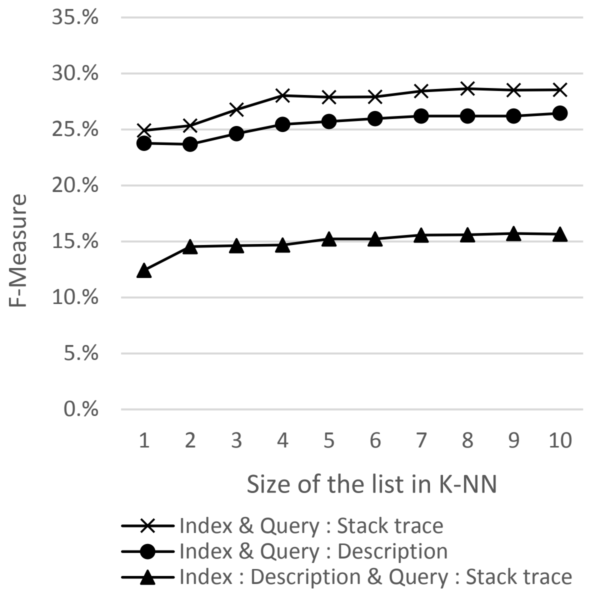

Figure 6. F-Measure of predicting Blocker severity with varying list size

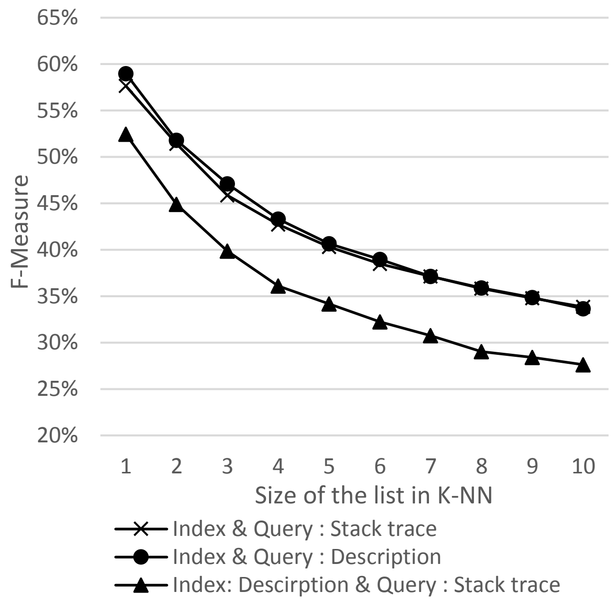

Figure 7. F-Measure of predicting Major severity with varying list size

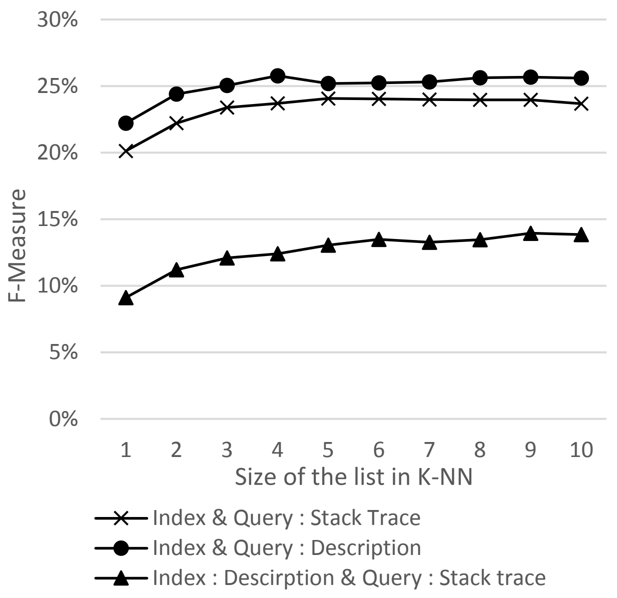

Figure 8. F-Measure of predicting Minor severity with varying list size

Figure 6 shows that similar to the case of Critical severity labels, when applying our approach to the Blocker severity labels, the severity prediction was best for the first scenario (i.e., when indexing function names of stack traces in the training set, and querying function names of new incoming bug report) and performed worse for the third scenario. This confirms our previous findings that the informal description in bug reports adds noise to the feature vector and deteriorates the prediction performance.

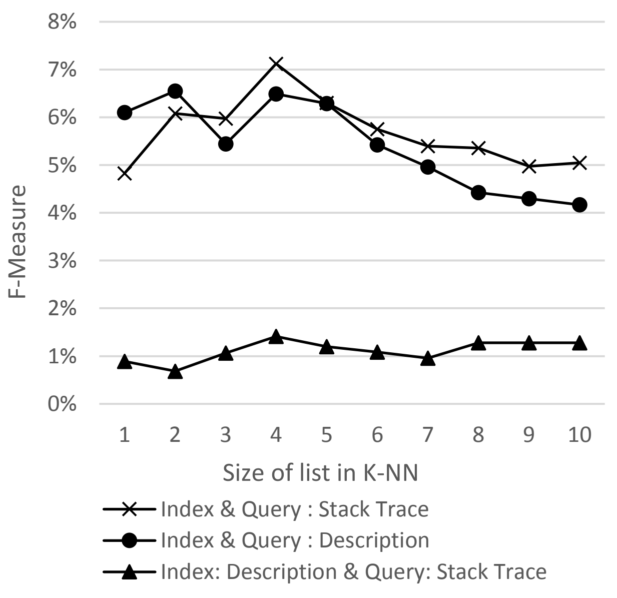

Figure 9. F-Measure of predicting Trivial severity with varying list size

Figure 7 shows that in the case of Major severity the prediction performance for both stack traces and descriptions were comparable. This is due to the unbalanced dataset. The number of instances with Major severity labels is more than the total number of instances of other severity labels. The excessive number of bugs with Major severity makes the approach biased toward the Major severity label even when using a cost-sensitive classifier. The precision, recall and the F-measure were the same using stack traces or descriptions. However, as expected, severity prediction has the worst performance in the third scenario. This is consistent with the result of Critical (Figure 5) and Blocker (Figure 6) severities.

According to Figure 8, for predicting Minor severity labels, the prediction performance was slightly worse when indexing and querying stack traces function names were used compared to when the descriptions of bug reports were used. To uncover the reason for this, we investigated the stack traces of bug reports with Minor severity. We noticed that Minor bugs usually happen in a rare workflow of the software with special parameters. Thus, unlike bugs with Critical or Blocker severity, where several customers may report the same bug, each Minor bug refers to a unique malfunction in the system that does not affect the execution of the software severely. Consequently, bugs with Minor severity are rarely reported by multiple customers and hence do not have similar reports, which affects the performance of the classifier. Moreover, when these bugs are reported, stack traces are not always included, which explains the poor prediction performance when using stack traces as compared to the descriptions. According to Figure 8, the result of the third scenario is consistent with previous results of other severity labels.

For the Trivial severity label, the proposed online severity prediction approach performs best when indexing function names of stack traces in the training set and querying function names of incoming bug stack trace (Figure 9). Similar to the result obtained for the Critical, Blocker and Minor severities, the third scenario has the worst performance.

To conclude, the findings are very promising since they suggest that stack traces perform better than the descriptions of bug reports when predicting severity, which answers RQ1. We showed also that stack traces are almost always better than the description of bug reports for predicting bug severity, which addresses RQ2.

4.5 Threats to Validity

In this study, we used the Eclipse dataset to evaluate our approach, although our approach performed well on Eclipse, we need to apply it to additional datasets.

In our dataset, the completeness of stack traces can be a threat to validity. On the ground that stack traces are copied by users who are not necessarily experienced, only a section of the traces that may seem to be important may have been copied in the description section of the bug report. Thus, relying on stack traces not provided by the system might have misled our approach at some point.

The misclassification cost associated to each severity label in this study is reciprocal to the proportion of that severity label in the dataset. The effect of using different misclassification costs for each severity on the performance of the approach must be studied.

5. RELATED WORK

Antoniol et al. [1] conducted one of the earliest studies in categorizing issue reports according to their degree of importance. They extracted 1800 issues reported to bug tracking systems of Mozilla, Eclipse and JBoss (600 reports were extracted from each bug tracking system). They revised each of those issues and labeled them manually to either corrective maintenance (bug) or other maintenance requests (non-bug). They used words in the description of bug reports as features. They used term frequency for weighing feature vectors corresponding to each bug. In addition to the words in the description, the author added the value of the severity field as an extra feature to the feature vector. The authors applied feature reduction techniques to reduce the number of features. The authors used diverse classification methods such as decision tree, logistic regression, and naïve Bayes to classify issues as bugs or non-bugs. They trained each of the classifiers using the top 20 or the top 50 features selected by the feature selection technique. The accuracy of their approach when applied to Mozilla issues having 20 features is 67% in the best case and by increasing the number of features up to 50, they obtained an accuracy of 77%. The accuracy of the approach when applied to Eclipse with 20 features is 81% and when having 50 features is 82%. The accuracy of the approach when applied to JBoss issues having 20 features is 80% and with 50 features is 82%. By visualizing the decision tree and showing the logistic regression coefficients, they concluded that some terms have more discriminative power than others. They concluded that as an example for Eclipse, a word like "Enhancement", extracted from severity, is a good indicator of a non-bug issue, while the word "Failure" is a good indicator of a bug. While the idea in this paper seems to be somewhat convincing, the main problem is that the authors used a set of manually labeled issues and added severity of an issue as a feature in the method. The authors ignored the original labeling of the bugs. The authors concluded that the value of the severity element is a good discriminator among bug and non-bug issues, which confirm that severity should have been used as a label, not a feature.

Menzies et al. [12] conducted a study on an industrial system used by NASA. NASA uses a bug tracking system called Project and Issue Tracking System (PITS). NASA uses a five-point scale from one to five (worst to dullest) to assign severity to the issues. The authors introduced a tool called SEVERIS that predicts the severity of an issue using text mining techniques applied to the description of the bugs. Considering that having all descriptions from all bugs creates a large number of features, the author used the TF-IDF score of each term to rank them. Then they cut all but the top K features. They also did another round of feature reduction using information gain applied to the top K features. They used rule learner to deduce rules from the weighted features. For the case study, they used five different datasets. The main problem with the datasets was that they did not have any bug with severity 1 (critical severity) and the total number of bugs was too small (only 3877 bugs). They calculated precision, recall and F-Measure to evaluate SEVERIS. Using the top 100 words as features, F-measure was in average 50%. They showed that, in the NASA dataset, using the top 3 features or the top 100 features does not change the F-measure significantly. This result shows that predicting severities using a small number of features with a good discriminative power is possible.

Lamkanfi et al. [7] conducted a study to calculate the accuracy of text mining techniques applied to the summary of bug reports to predict their severity. The authors used the dataset of open source software such as Mozilla, Eclipse and Gnome to evaluate their approach. They modeled the bug severity prediction problem as a document classification problem. They used a Naïve Bayes classifier for the classification purpose, which is based on the presence or absence of a word (feature) in the summary or the description of bug reports. They preprocessed the bug reports summary by tokenization, stop-word removal and stemming. They used the summary or the description of bug reports for evaluation. The authors organized the bugs according to the product and components that are affected. Using the summary of the bug reports and product component specific analysis they reported precision and recall of at least 70% for all datasets. They did a study on the most important terms that are used as features in their approach. They reported that words like "crash" or "memory" are good indicators of severe bugs, but words like "typo" are good indicators of non-bugs. The authors did a second round of experiments using only descriptions instead of summaries and showed that using descriptions decreases the performance of the classifier. They run experiments with different sizes of training sets to show the effect of the training set size on the result. They concluded that a training set having 500 bugs is enough to have a generalizable result for all datasets. In another round of experiments, they showed that isolating bugs according to the affected components and products when applying a severity prediction approach to Eclipse bug tracking system shows better results than cross component analysis.

Lamkanfi et al. [8] compared the result of diverse mining algorithms applied to bug repositories to predict the coarse grain severity of the bugs. Eclipse and Gnome are the datasets that are used to evaluate the accuracy. They extracted words as feature from the description of bug reports in the datasets. They extracted bug reports and categorized them according to their product and component. They used Naive Bayes, Naive Bayes multinomial, 1-Nearest Neighbor and Support Vector machine classifiers to predict severity of the bugs. Due to the fact that different classification approaches need different ways to weigh feature vectors, they weighed feature vectors according to presence or absence of each term when using Naïve Bayes classifier. They used term frequency to weigh feature vectors when using Naïve Bayes Multinomial. They also used term frequency and inverse document frequency (TF-IDF) to weigh the feature vectors when using 1-Nearest Neighbor or Support Vector Machine. They calculated precision, recall and the area under curve (AUC) of coarse grain severity prediction approach for each dataset. They showed that using Naïve Byes Multinomial, according to the area under curve, the accuracy reaches 80%, which is higher than other approaches. Furthermore, they showed that using Naïve Bayes classifier, a stable accuracy value is achieved having 250 bug reports of each severity type for training. Thus, increasing the training set by adding more than 250 bug reports does not change the accuracy in the studied datasets.

Yang et al. [18] studied the benefit of feature selection in the coarse grain severity prediction process. They compared severity prediction capability of Naïve Bayes classifier after applying diverse feature selection techniques. They used three different feature selection techniques including information gain, Chi-square and correlation coefficient. They used Eclipse and Mozilla datasets. The main goal in information gain is to find a feature that can represent a category. High information gain can show that a bug, which contains the feature, is severe. A low information gain can show that bug is non-severe. Chi-square is a score similar among severe as well as non-severe bugs. The correlation coefficient shows polarity of a term in a category. They used true positive rate (TPR), false positive rate (FPR) and area under curve (AUC) to evaluate the performance of each feature selection technique. They extracted the top M features each time and did classification based on those features. They concluded that the best feature selection technique for Eclipse and Mozilla is the correlation coefficient.

Coarse grain severity prediction categorizes reported issues as either severe or non-severe. Thus, it gives the triager a general idea about the severity of the bug report. However, bug tracking systems follow a finer grain severity categorization. A fine grain severity categorization gives triager more flexibility for deciding on how to deal with the reported bug. While all of the previous studies concentrated on a coarse-grain severity prediction, Tian et al. [16] did a study on a finer grain severity prediction approach. For a fine grain severity prediction, having features with more discriminability power is a necessity. Thus, the authors refined the features to increase their discriminability power. They used Open Office, Mozilla and Eclipse to build the dataset. Since bug reports with normal severity are usually chosen arbitrarily, the authors removed them from the datasets. Furthermore, they removed bug reports with enhancement severity because they are also not considered as bugs. They used the K-nearest neighbor to predict severity of each bug report in the test set. They chose the K-nearest neighbor because bugs that are similar to each other are expected to have the same severity. They applied the K-nearest neighbor to each bug in the test set, and then the severity of that bug is predicted according to the severity of the returned similar bug reports. They used the textual description as well as categorical fields of bug reports to calculate the similarity of bug reports. They used unigram and bigram features, extracted from the summary and the description of bug reports as the textual features and product and component in the bug report as categorical features. The similarity of unigram and bigrams textual features is calculated with an extended version of BM25 called $BM25_{ext}$ and similarity of categorical features is a binary value based on their equality. They used a linear combination of these four features to calculate similarity of bug reports. They showed that their approach outperforms SEVERIS.

Yang et al. [19] studied the influence of four quality indicators of bug reports in severity prediction. They studied the coarse grain severity prediction using a naïve Bayes classifier. They used bug reports from 2 components of Eclipse. They studied usefulness of four quality indicators: stack traces, report length, attachment and step to reproduce. They did two series of experiments to measure the impact of quality indicators on predicting severity of bug report. In the first round of experiments, they studied only the existence of each quality indicator. In the second round of experiments, the authors studied the effect of quantitative values of quality indicators in predicting severity. They extracted these quantitative values from each quality indicators differently. Examples of indicators are the number of functions in stack traces, the number of attachments in a bug report, the length of the report, etc. They concluded that among all these indicators, stack traces are the best for predicting severity. The authors, however, did not experiment with the content of stack traces, which is the objective of our approach. Their result encouraged us to study the severity prediction using the content of stack traces of bug reports.

Yang et al. [19] used topic modeling techniques to further improve the fine grain severity prediction methods. They only compared the test bug report with the historical bugs if they had the same product, components, and priority. The authors used LDA technique for extracting topics from corpus of documents. Each topic is considered as a bag of words. They used smoothed unigram vectors instead of vector space models and used KL divergence as the metric to measure similarity of smoothed vectors. They used Mozilla, Eclipse and NetBeans datasets to evaluate their approach. They implemented an online approach in which with each incoming bug report its probability vector is extracted. Then, according to its topic, historical bug reports with the same topic are extracted. The K-nearest neighbor is then applied to the bug with the same topic and a list of similar bugs is returned. Then according to the labels of bugs in the returned list, the severity of the incoming bug is predicted.

Bhattacharya et al. [2] proposed a graph-based analysis of software systems to explore alternate avenues in predicting severity of bug reports. They used Firefox, Eclipse and MySQL datasets to evaluate their approach. A graph can be built based on different aspects of the software system. They built source code graphs based on function calls (i.e., a static call graph) or modules (module collaboration graph) and, in a more abstract level, they built a graph based on developer's collaboration. They showed that graph-based metrics such as average degree, clustering coefficient, node rank, graph diameter and assortativity can be used to characterize software structure and evolution. They used node rank among all of the graph metrics to specify function or modules which if buggy can cause high severe bugs. They concluded that node rank based on function call graph is a good indicator of bug severity. They also did further studies on Modularity Ratio and found that it can be used as a good indicator for finding modules that need less maintenance effort.

Zhang et al. [20] used concept profile of the bug repositories of Eclipse and Mozilla to predict the severity of the incoming bug report. They used 90% of instances in dataset for training and the remaining 10% of the datasets for testing to evaluate their severity prediction approach. They extracted concept terms for each fine grain severity label in the training set. They calculated concept terms threshold. Then, they built concept profile corresponding to each severity label using concept terms. Similar to Yang et al. [17], instead of using vector space models, they represented each bug and concept profile using a probability vector and used KL divergence for calculating similarity between each bug in the testing set and each concept profile which corresponds to each severity label in the training set. It is shown that the proposed approach can be used for predicting severity and furthermore has better performance compared to other machine learning algorithms.

There exist other studies that leverage the use of stack traces to improve the process of bug handling including the detection of duplicate bug reports [4] and bug reproduction [13].

6. CONCLUSION AND FUTURE WORK

In this paper, we studied the capability of stack traces to predict the severity of bug reports. To the best of our knowledge, this is the first study that explores the use of stack traces for predicting the severity of bug reports. We showed that using stack traces, the severity of a bug report could be predicted with higher accuracy compared to using the description of a bug report. Furthermore, using stack traces significantly decreases the number of features used for predicting the bug severity.

Unfortunately, not all bug reports have stack traces. This makes the distribution of severity labels unbalanced. An unbalanced dataset decreases the classification accuracy for minority labels. To deal with this problem, we used cost effective classification to improve the classification accuracy of minority labels.

In the future, we are planning to extend this study by applying it to more bug repositories. Furthermore, we will study the effect of associating different misclassification costs to each severity label in the prediction process.

7. ACKNOWLEDGEMENT

This research is partly supported by a grant from the Natural Science and Engineering Research Council (NSERC) of Canada.

8. 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 requests," In *Proc. of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08),* pp. 304–318, 2008.

[2]. P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," In *Proc. of the 34th International Conference on Software Engineering (ICSE'12),* pp. 419– 429, 2012.

[3]. J. L. Davidson, N. Mohan, and C. Jensen, "Coping with duplicate bug reports in free/open source software projects," In *Proc. of IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC'11),* pp. 101–108, 2011.

[4]. N. Ebrahimi, A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata," In *Proc. of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON'15),* pp. 201-210, 2015.

[5]. H. V. Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," In *Proc. of the 11th Working Conference on Mining Software Repositories (MSR'14),* pp. 72–81, 2014.

[6]. J. Gou, L. Du, Y. Zhang, T. Xiong, "A New Distance-weighted K-nearest Neighbor Classifier," *In Journal of Information & Computational Science, 9(6),* pp. 1429-1436, 2012.

[7]. A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories (MSR'10),* pp. 1–10, 2010.

[8]. A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," In *Proc. of 15th European Conference on Software Maintenance and Reengineering (CSMR'11),* pp. 249–258, 2011.

[9]. J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," In *Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13),* pp. 69–78, 2013.

[10]. A. Maiga, A. Hamou-Lhadj, M. Nayrolles, K. Sabor, A. Larsson, "An Empirical Study on the Handling of Crash Reports in a Large Software Company: An Experience Report," In *Proc. of the 31st International Conference on Software Maintenance and Evolution (ICSME'15),* pp. 342-351, 2015.

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

[12]. T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," In *Proc. of the International Conference on Software Maintenance (ICSM'08),* pp. 346–355, 2008.

[13]. M. Nayrolles, A. Hamou-Lhadj, S. Tahar and A. Larsson, "JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," In *Proc. of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (a merger of WCRE and CSMR) (SANER'15),* pp. 101–110, 2015.

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

[15]. Z. Qin, A. T. Wang, C. Zhang, and S. Zhang, "Cost-Sensitive Classification with k Nearest Neighbors," In *Proc. of the 6th International Conference on Knowledge Science, Engineering and Management (KSEM'13),* pp. 112–131, 2013.

[16]. Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," In *Proc. of the 19th Working Conference on Reverse Engineering,* pp. 215–224, 2012.

[17]. 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 *Proc. of the 38th Annual Computer Software and Applications Conference (COMPSAC'14),* pp. 97–106, 2014.

[18]. C.-Z. Yang, C-C. Hou, W-C. Kao, and I.-X. Chen, "An empirical study on improving severity prediction of defect reports using feature selection," In *Proc. of the 19th Asia-Pacific Software Engineering Conference (APSEC '12),* pp. 240–249, 2012.

[19]. Z. Yang, K. Y. Chen, W. C. Kao, and C. C. Yang, "Improving severity prediction on software bug reports using quality indicators," In *Proc. of the 5th IEEE International Conference on Software Engineering and Service Science (ICSESS'14),* pp. 216–219, 2014.

[20]. T. Zhang, G. Yang, B. Lee, and A. T. S. Chan, "Predicting severity of bug report by mining bug repository with concept profile," In *Proc. of the 30th Annual ACM Symposium on Applied Computing (SAC'15),* pp. 1553–1558, 2015.

[21]. J. Zhang, I. Mani, "KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction," In *Proc. of the International Conference on Machine Learning (ICML'03), Workshop on Learning from Imbalanced Data Sets,* 2003.