Automatic Prediction of the Severity of Bugs Using Stack Traces and Categorical Features

Korosh Koochekian Sabor[1], Mohammad Hamdaqa[2], Abdelwahab Hamou-Lhadj[1]
[1]Department of Electrical and Computer Engineering
Concordia University, Montréal, QC, Canada
{k_kooche, abdelw}@ece.concordia.ca
[2]School of Computer Science
Reykjavik University, Iceland
mhamdaqa@ru.is

Abstract

Context: The severity of a bug is often used as an indicator of the impact of a bug on system functionality. It is often used by developers to prioritize bugs which need to be fixed sooner than others. The problem is that, for various reasons, bug report submitters often enter an incorrect severity level, delaying the bug resolution process. Techniques that can automatically predict the severity of a bug may significantly reduce the bug triaging overhead. In our previous work, we showed that the stack traces can be used for predicting the severity of a bug with a reasonable accuracy.

Objective: In this study, we expand our previous work by proposing an approach for predicting the severity of a bug by combining bug report categorical features and stack traces. We focus on three categorical features, namely, the faulty product, faulty component, and operating system. A software system is composed of many products; each has a set of components. Components interact with each to provide the functionality of the product. The operating system field refers to the operating system on which the software was running on during the crash.

Method: The proposed approach uses a linear combination of stack trace and categorical features similarity to predict the severity of bug reports. We adopt a cost sensitive K Nearest Neighbor approach to overcome the unbalance label distribution problem and improve the classifier accuracy.

Results: Our experiments on bug reports of the Eclipse project submitted between 2001 and 2015 and Gnome submitted between 1999 and 2015 show that the use of categorical features improves the prediction accuracy of an approach that uses stack traces alone by 5% to 20%.

Conclusion: The accuracy of predicting the severity of bugs is higher when combining stack traces and the three categorical features: product, component, and operating system.

Keywords— Bug Severity; Stack Traces; Machine Learning; Mining Software Repositories; Software Maintenance;

1. Introduction

Software defects that go undetected during the verification phases often cause system crashes and other undesirable behaviors. When a crash occurs, users have the option to report the crash using bug tracking systems such as Windows Error Reporting tool[1], Mozilla crash reporting system[2], and Ubuntu's Apport crash reporting tool[3]. The main objective of these systems is to enable end users to submit crash/bug reports, where they can report various information about a bug including a textual description, the stack trace associated with the bug, and other categorical features such as the perceived severity of the bug and its operating environment (e.g., product, component and operating system).

After a bug report is submitted, a team of triagers examine each report in order to redirect the ones requiring fixes to the developers of the system. For software systems with a large client base like Eclipse, triaging tends to be a tedious and time-consuming task. This is due to the large number of reports received [DMJ11]. Triagers usually prioritize the bug reports using typically the reported *bug severity*. A bug severity is defined as a measure of how a defect affects the normal functionality of the system [LDSV11, YHKC12].

Despite the existence of guidelines on how to determine the severity level of a bug, studies have shown that users often specify the severity level incorrectly, mainly because they lack expertise in the system domain [ZCYLL16, MHNSL15]. An inaccurate severity level may cause delays in processing bug reports as shown by Zhang et al. [ZCYLL16]. Triagers often have to review bug reports in order to determine the right severity level, a task that is usually achieved manually, hindering productivity and slowing down the bug resolution process [YHKC12, TLS12, YZL14, ZYLC15].

To address this issue, several methods [LDSV11, AADPKG08, MM08] have been proposed to predict the correct severity level of a bug. These techniques treat the problem as a classification problem, by learning from historical bug reports in order to predict the severity levels of the incoming ones. These studies mainly use words in bug report descriptions as the main features for classification. In our previous work [SHH16], we showed that using the information in stack traces, which are attached to bug reports, the severity of a bug could be predicted more accurately compared to using bug report descriptions.

[1]http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx
[2]http://crash-stats.mozilla.com
[3]https://wiki.ubuntu.com/Apport

In this study, we improve our previous work by adding categorical information of bug reports. Categorical information of a bug report provides information about the environment in which the bug has been discovered. Such information is important when assigning the bug severity, particularly for system and integration bugs, which are the most difficult ones to identify at development time [MM08]. For this reason, in this work, we extend our previous work by proposing a severity prediction approach that uses the combination of stack traces and categorical features of bug reports. We focus on three categorical features, namely the faulty product, component, and operating system.

To the best of our knowledge, this is the first time that stack traces with categorical features are used to predict the severity of bugs. We evaluate our approach on 11,825 and 153,343 bug reports from Eclipse and Gnome bug repositories. We show that by combining stack traces and categorical features, the prediction accuracy of our previous approach that uses solely stack traces can be improved by 5% for Eclipse and 20% for Gnome.

Several original contributions to the literature emerge from our research.

- Devising a new approach for predicting the severity of bugs that combines stack traces and bug report categorical features.
- Comparing the results of the new approach to the state of the art in this field.

The remaining parts of this paper are organized as follows: Section 2, lists the main features of a bug report. Section 3 presents our severity prediction approach, followed by evaluation and experimental results in Section 4. The limitations and threats to validity are presented in Section 5. Section 6 presents the background and related work. Finally, we conclude the paper in Section 7.

2. Bug Report Features

A bug report consists of a description, a stack trace (optional), and other categorical features.

2.1. Bug Report Description

A bug report description is entered by bug report submitters to explain the observed behaviour during the crash. A good description should provide enough information to guide a developer in understanding what user or tester has observed during the crash. Bug descriptions are written in natural language and can be quite verbose. An example of a well written description is "Download fails on Customer Report when user clicks on the downed button". An example of an ineffective description is "Download does not work" as it does not provide details about when or how this failure occurs. The quality of a bug report description

depends on who is writing the description and hence it can be subjective. In addition, bug report descriptions are subject to the imprecision and ambiguity of natural language.

2.2. Stack Traces

A stack trace is a sequence of function calls that are in the memory stack when the crash occurs. An example of a stack trace is shown in Figure 1. This trace is taken from the Eclipse bug repository. It represents the stack trace of Bug 38601. The bug was caused by a failure of checking for a null pointer in the search for a method reference in the Eclipse.

```
5- org.eclipse.jdt.internal.corext.util.Strings.convertIntoLines()
4- org.eclipse.jdt.internal.ui.text.java.hover.JavaSourceHover.getHoverInfo()
3-  org.eclipse.jdt.internal.ui.text.java.hover.AbstractJavaEditorTextHover.getHoverInfo()
2- org.eclipse.jdt.internal.ui.text.java.hover.JavaEditorTextHoverProxy.getHoverInfo()
1- org.eclipse.jface.text.TextViewerHoverManager.run()
```

Figure 1. The stack trace for bug 38601 from Eclipse bug repository

Bettenburg et al. showed that stack traces are considered to be very by developers in order to fix a bug [BJSWPZ08]. Nayrolles et al. showed that stack traces are not only useful for understanding a bug by also for bug reproduction [NHTL15][NHTL16]. Stack traces have also been used to detect duplicate bug reports in large bug report repositories [ETIHK19, EH15, EIHH16]. Currently in the Bugzilla bug tracking system, attaching a stack trace is not mandatory and not all bug reports have stack traces. Both Eclipse and Gnome use the Bugzilla bug tracking system, so there is no special section for stack traces and it is up to users to copy a stack trace into the bug report description field.

2.3. Categorical Features

In addition to the description of the bug and a stack trace, users must choose other categorical fields, which can further help developers to fix the bug. Categorical features of a bug include the product, component, version, operating system, severity of the bug, submitter’s name, etc. The triaging team reviews bug report categorical fields may decide to update them based on their experience and the information provided.

In this study, we choose to use, in addition to stack traces, three categorical features to predict the severity of a bug: the product, the component, and the operating system. Severity of a bug is typically related to the product and component in which the bug occurs. Bugs that are discovered in core products and components should receive immediate attention to reduce system downtimes. These bugs may be categorized as highly

severe. It is therefore intuitive to select product and component fields when predicting the severity of bugs. We also decided to experiment with other categorical features such as the operating system, the software version, the submitter, etc. We proceeded using the forward selection approach presented by Ian et al [WFH11]. The idea is to add each feature one at the time and see the effect on accuracy. In this study, we found that, except for product, component, and operating system, the other features did not improve accuracy. For this reason, we settled on product, component, and operating system as our final set of categorical features.

2.3.1. Levels of Bug Severity

Severity can be manually assigned by a user and describes the level of the impact that a bug has on the system. The followings are the main severity levels of Eclipse Bugzilla[1]:

- Blocker severity shows bugs that halt the development process. Blocking bugs are the bugs that do not have any work around.
- Critical bugs are bugs that can cause loss of data or sever memory leaks.
- Major bugs are the bugs that are seriously obstacle to work with the software system.
- Normal bugs are chosen when the user is not sure about the bug or if the bug is related to documentation.
- Minor bugs are the ones which are worth reporting but do not interfere with the functionality of the system.
- Trivial bugs are cosmetic bugs such as typos in the java docs.

There are also bug reports, which are marked as enhancements. These reports do not describe defects in the system but rather possible enhancements to the system such as adding new features or refactoring opportunities to improve system quality.

3. The Proposed Approach

In this paper, we propose a new bug severity prediction approach that uses stack traces and categorical features to predict the bug severity. We achieve this by calculating the similarity of each incoming bug report to all the previous bug reports in the bug tracking system. Similarity is calculated based on a linear combination of the similarity of the bug report stack traces and that of their categorical features, namely

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

product, component and operating system. The severity of the bug is then chosen based on the severity of the K nearest neighbor (K nearest bug reports) to the incoming bug. Figure 4 shows the overall approach.

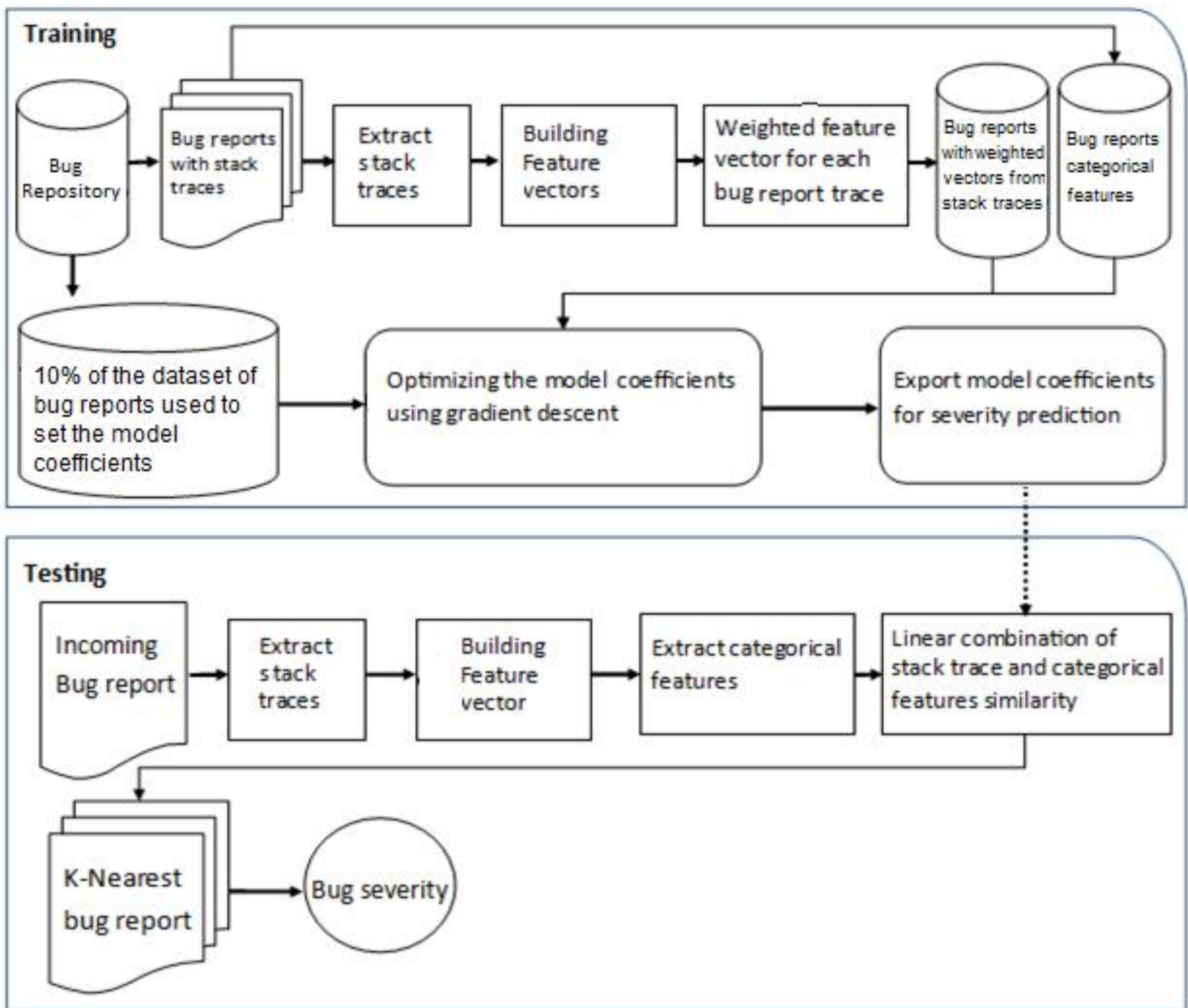

Figure 4. Overall approach

3.1 Extracting Features from Bug Reports

In Bugzilla, users copy the stack trace of a bug inside the description of the bug report. To identify and extract these stack traces, we use the regular expression of Figure 2, from the work of Lerch et al. [LM13]. Moreover, to extract stack traces form the Gnome bug repository, we defined and implemented the regular expression presented in Figure 3.

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

Figure 2. Regular expression used for extracting stack traces from bug report description in the Eclipse bug repository [LM13].

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

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

Categorical features can be found in the XML preview of a bug report. Each categorical feature is enclosed by a specific XML tag that conforms to the Bugzilla schema. Since both Gnome and Eclipse use the same bug tracking system, the XML preview is structured in the same format. In our work, we implemented a custom parser to extract this information from the bug report XML previews.

3.2 Measuring Similarity between Bug Reports

Given two bug reports (B_1, B_2), the combined similarity is calculated as follows:

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

where $feature_1, feature_2$, $feature_3$ and $feature_4$ are defined as follows:

$$feature_1 = Similarity\ of\ stack\ traces$$

$$feature_2 = \begin{cases} 1 & if\ B1.Product = B2.Product \\ 0 & otherwise \end{cases}$$

$$feature_3 = \begin{cases} 1 & if\ B1.Component = B2.Component \\ 0 & otherwise \end{cases}$$

$$feature_4 = \begin{cases} 1 & if\ B1.operating\ system = B2.operating\ system \\ 0 & otherwise \end{cases}$$

Based on Equation (1), the similarity of two bug reports is the linear combination of their corresponding stack traces and categorical features similarities. We discuss the similarity of stack traces in the next section. The similarity of categorical features is one if they are the same and zero if they are not. Note that Equation (1) uses four parameters $\mathrm{w_i}$: $(\mathrm{w_1}, \mathrm{w_2}, \mathrm{w_3}, \mathrm{w_4})$, which represent the weight of each feature. These parameters need to be optimized to reflect the importance of each feature. To this end, we use an adaptive learning approach that relies on a cost function and gradient descent in a similar way to the one proposed by Sun et al. [SLKJ11]. The format of our training set is similar to the one used by Sun et al. [SLKJ11]. The dataset contains triples in the form of (*q*, *rel*, *irr*) where q is the incoming bug report, *rel* is a bug report with the same severity and *irr* is a bug report with a different severity. The method used to create the training set (TS) is shown in Algorithm 1.

Based on Algorithm 1, in the first step, bug reports are grouped based on their severity levels. Next, for each bug report (*q*) in each severity group we need to find bug reports that have the same severity level and those with a different severity. We chose another bug report from the same severity group (*rel*) and chose a bug report from another severity group (*irr*). The process continues until we create all (*rel*, *irr*) pairs for the

bug report (*q*). We continue the same steps for all the bug reports in our training set to create the (*q, rel, irr*) triples. For example, assume we have two severity labels $\{S_1, S_2\}$ and Bug_1 and Bug_2 have severity level S_1, and Bug_3 and Bug_4 have severity level S_2. The triples with the format (*q, rel, irr*) for training in this case is: (bug_1, bug_2, bug_3), (bug_2, bug_1, bug_4), (bug_3, bug_4, bug_1), (bug_4, bug_3, bug_2).

```
TS = ∅: training set
N>0 parameter controlling size of TS
G={G_1, G_2, G_3, G_4, … , G_n }
G_i = bug reports of severity i
----------------------------------------
For each Group G_i in the repository do
  R= all bug reports in Group G_i
  For each report q in R do
     For each report rel in R-{q} do
        For i=1 to N do
            Randomly choose a report irr out of R
            TS=TS U {(q,rel,irr)}
        End for
     End for
  End For
End for
Return TS
```

Algorithm 1. Creating training set.

To define a cost function based on the created triples in order to optimize the free parameters (w_1, w_2, w_3, w_4), we use the cost function, RankNet, that is defined in [SLKJ11]:

$$Y = Sim(irr,q) - Sim(rel,q) \quad (2)$$

$$\text{RankNet}(I) = Log(1 + e^{Y}) \quad (3)$$

The goal is to minimize the defined RankNet cost function. The cost function is minimized when the similarity of bug reports with the same severity defined in Equation (1) is maximized and the similarity of the bug reports with different severities is minimized. To minimize the above cost function, we use a gradient descent algorithm as shown in Algorithm 2.

TS = ∅*: training set* *N>0 size of TS* *Ƞ: the tuning rate*

For *n=1 to N* **do**
  For each instance *I* in *TS in random order* **do**
    For each free parameter x in *sim()* **do**
      x=x- $Ƞ * \frac{\partial \text{RankNet}}{\partial x}$
    End for
  End for
End For

Algorithm 2. Optimization using gradient descent [SLKJ11].

3.3 Stack Trace Similarity

To compare two stack traces, we first create feature vectors using all distinct functions in all the stack traces, then weigh the feature vector for each stack trace using term-frequency and inverse document frequency (TF-IDF). We use cosine similarity to compare stack traces using their corresponding weighted feature vectors. We further elaborate on each of these steps in the following subsections

3.3.1 Building Feature Vectors

In our approach the term concept refers to a function name and a document refers to a stack trace. Before explaining the process, let us consider the following definitions:

- Let $T = f_{1,} f_{2,} \ldots\ldots, f_L$ be a stack trace of a bug that is generated by a crash in the system. *T* is a set function calls $f_{1,}$ to f_L, where *L* is the length of *T*.
- Let *Σ* be an alphabet of size *m* = *|Σ|* that represents distinct function names (terms) in the system.
- Let $\Gamma = T_{1,} T_{2,} \ldots\ldots, T_K$ be a collection of *K* traces that are generated by the process (or the system) and then provided for designing the severity prediction system.
- Each stack trace $T \in \Gamma$ could be mapped into a vector of size m functions, $T \rightarrow \emptyset(T)_{fi\epsilon\,\Sigma}$, in which each function name $f_i \in \Sigma$ either has the value one, which indicates the presence of the function or zero, which indicates the absence of that function.

Given the previous definitions, the term-vector can be weighted by the term-frequency (tf) as shown in Equation 4:

$$\phi_{tf}(f,T) = freq(f_i); i = 1, \dots, m \qquad (4)$$

In Equation (4), *freq(fi)* is the number of times the function f_i appears in T normalized by L, where L is total number of functions calls in *T*.

Term frequency only shows the local importance of a function f_i in a stack trace. However, unique function names (terms) that appear frequently in a small number of stack traces convey more information than those that are frequent in all stack traces. Hence, the inverse document frequency (idf) is used to adjust the weights of functions according to their presence across all stack traces. The term vector weighted by the tf.idf is therefore given by Equation (5):

$$\phi_{tf.idf}(f,T,\Gamma) = \frac{\mathrm{K}}{df(f_i)} freq(f_i); i = 1, \dots, m \qquad (5)$$

In Equation (5), $df(f_i)$ is the number of traces in the collection Γ that contains the function name f_i. Thus, higher score is given to functions that are very common in a stack trace, but rare in other traces of the collection Γ.

Next, for each stack trace $\mathrm{T}_{\mathrm{i},} \in \Gamma$ a feature vector based on the model is constructed and weighed using TF-IDF. The output of this part is an adjacency matrix, where each row shows a weighed stack trace corresponding to a bug.

3.3.2 Measuring Similarity between Stack Traces

To compare two stack traces, we measure the distance between their corresponding feature vectors using the cosine similarity measure. Given two vectors $V_1 =< v_{11}, v_{12}, \dots.. v_{1n} >$ and $V_2 =< v_{21}, v_{22}, \dots.. v_{2n}>$, the cosine similarity is calculated as follows [MRS08]:

$$Cos(\theta) = \frac{V_1.V_2}{|V_1|.|V_2|} \qquad (6)$$

As shown in Equation (6), the cosine similarity between the two vectors is the cosine of the angle between the two vectors. It is equal to the dot product of the two vectors divided by the multiplication of their sizes.

3.4 KNN Classifier for Severity Classification

After calculating the similarity of bug reports by a linear combination of their stack traces and categorical features similarity, we use KNN to predict the severity of the incoming bug report. KNN is an instance-based lazy learning algorithm [TLS12]. Given a feature vector, KNN returns the K most similar instances to that vector. Following a typical KNN classification algorithm, in our case, KNN has two phases: in the first phase, the similarity of the incoming bug report B_i to all the bugs in the training set is calculated.

Accordingly, based on the value of (K), which is a constant value that defines the number of returned neighbors, the algorithm returns the K nearest relevant instances. In the second phase, a voting algorithm (e.g., majority voting) among the labels of the k most similar instances is used to classify the incoming bug report B_i. The label of instance X (i.e., the instance that corresponds to B_i) can be determined given the labels set C by majority voting as in Equation (7) [PF13].

$$C(X) = \underset{c_j \in C}{argmax} \quad score\,(c_j, neighbors_k(X)) \quad (7)$$

In Equation (7) $neighbors_k(X)$ is the K nearest neighbors of instance X, argmax returns the label that maximizes the score function, which is defined in Equation (8) [PF13].

$$Score\,(c_j, N) = \sum_{Y \in N} [class(y) = c_j] \quad (8)$$

In Equation (8), $class(y) = c_j$ returns either one or zero. It is one when $class(y) = c_j$ and zero otherwise. In Equation (8) the frequency of appearance of a label is the only factor that determines the output label. Finally, the label with the highest frequency among K labels is chosen as the label of the incoming bug report.

While the score function in Equation (8) shows promising results, to further enhance the prediction capability of our approach, we also need to consider the similarity of each top K returned bug reports by giving more weights to the labels of the bug reports that are closer to the incoming bug. The reason is that, bug reports that are closer to the incoming bug report B_i will most probably have the same label as B_i.

If we assume the distance of the closest bug report in the sorted list as $dist_1$ and the distance of the farthest bug report in the retrieved list as $dist_k$, then the weight of each label in the list can be calculated as Equation (9), as discussed by Gou et al. [GDZX12], where $dist_i$ is the distance of bug report i.

$$w_i = \begin{cases} \frac{dist_k - dist_i}{dist_k - dist_1}, & if\ dist_k \neq dist_1 \\ 1, & if\ dist_k = dist_1 \end{cases} \quad (9)$$

Equation (9) ensures that higher weights are given to bug reports that are closer to the incoming bug report. We need to update the score function in Equation (8) to incorporate the weights calculated in Equation (9). The updated score function is shown in Equation (10) [PF13]:

$$\text{Score}\ (c_j, N) = \sum_{Y \in N} w(x,y) \times [class(y) = c_j] \quad (10)$$

where *w(x,y)* is the weight of each instance in the top k similar returned instances which is calculated by Equation (9) according to its distance to the incoming bug report B_i corresponding to instance *X*. After calculating the weight of each label, according to Equation (10), the label with the highest weight is selected as the output label.

4. Evaluation

The goal of this section is to evaluate the accuracy of predicting the severity of bugs using stack traces and categorical features compared to our previous approach that only uses stack traces and the approach that uses bug report descriptions. More precisely, the experiment aims to answer the following questions:

RQ1. How much improvement (if any) could be obtained by using stack traces over the approach that uses bug report descriptions and a random approach?

RQ2. Using the KNN classifier for predicting bug severity, how different the number of neighbors can affect the F-measure of the proposed approach?

RQ3. What is the impact of adding categorical features to stack traces on accuracy of the severity prediction compared to solely using stack trace information and a random approach?

Answering RQ1 shows the importance of using stack traces as an alternative to bug report descriptions. Answering RQ2 shows the sensitivity of the proposed approach based on the selected number of nearest neighbors. Answering RQ3 assesses the added value of categorical features.

4.1. Experimental Setup

4.1.1 The Datasets

The datasets used in this paper consist of bug reports from the Eclipse and Gnome bug repositories. Both datasets are used extensively in similar studies [YZL14, BINF12, ZYLC15, SHH16]. We downloaded the Eclipse bug reports that were reported from October 2001 to February 2015 using a script that uses the bug report URLs. For example, URL https://bugs.eclipse.org/bugs/show_bug.cgi?id=1 refers to Bug Report #1. To obtain the next report, we simply need to change the id to 2, etc. In total, we collected of 455,700 bug reports by varying the bug report id until we reach bug reports that were reported in February 2015. After processing the collected bug reports, we found that about 1,000 reports were marked invalid. These bug reports may have been submitted, but later removed by the triaging teams, perhaps for maintenance purposes. We removed these reports from our dataset. In addition, we found that 297,151 have normal

severity and 66,873 are labeled as enhancements. We also removed these reports from the dataset, resulting in a total of 90,676 valid bug reports with severities other than normal or enhancement. After applying the regular expression, we have found that 11,925 bug reports have at least one stack trace in the description. Stack traces with less than three functions were removed since they are partial stack traces and may mislead the approach. The resulting Eclipse dataset contains a total of 11,825 bug reports with17,695 stack traces in their description (see Table 1).

Table 1. Characteristics of the datasets

Data	**Eclipse**	**Gnome**
Total number of reports	455,700	752,300
Total number of enhancement reports	66,873	57,446
Total number of bug reports with normal severity	297,151	297,831
Total number of bug reports removed from the dataset	1,000	54,500
Total number of BRs excluding normal and Enhancement	90,676	342,523
Total number of BRs with stack traces	**11,925**	**153,343**

The Gnome dataset used in this paper contains bug reports from February 1997 to August 2015. The dataset contains 752,300 reports out of which 57,446 are labeled as enhancement and 297,831 are labeled as normal. Similar to Eclipse, we found that 54,500 bug report were marked invalid. We removed these reports from our dataset (see Table 1). From the remaining 342,523 bugs, 153,385 bug reports come with one or more stack traces in their description. After eliminating bug reports with partial stack traces, we were left with a total of 153,343 bug reports with at least one stack trace in their description.

In these repositories, stack traces are embedded in the bug report descriptions. As explained earlier, to extract the content of the stack traces in Eclipse, we use the same regular expression presented by Lerch et al. [LM13], and for Gnome, we use the regular expression shown in Figure 3.

For Eclipse, the extracted stack traces are preprocessed to remove noise in the data such as native methods, used when a call to the Java library is performed. There are also lines in the traces that are labelled ‘unknown source’. This occurs due to the way debugging parameters were set.

Normal severity is the default choice when submitting a bug report, so it is usually chosen arbitrarily. Consistent with previous severity prediction studies (e.g. [LDSV11, AADPKG08, MM08]), we removed

bug reports with normal severity. Furthermore, enhancements are not considered as bugs, and are also removed from the datasets when performing the experiments [SHH16].

Consistent with the previous studies (e.g., [TLS12, YZL14, ZYLC15]), we use the severity of bugs from the bug tracking system as label. These severities are set by users and then further adjusted by bug triagers. Figure 5 shows the process of assessing bug report severities by triagers after the bug is submitted by the user.

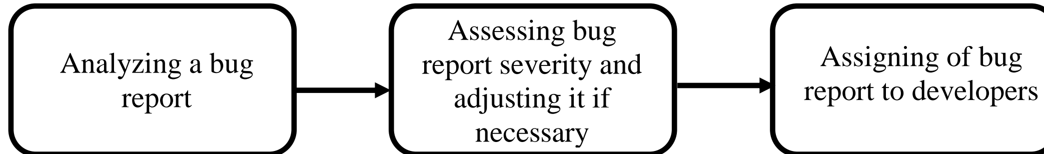

Figure 5. The bug report handling process (adapted from [SSG15])

4.1.2 The Training and Testing Process

In our approach, for each dataset, we sort bug reports by their creation date and use the first 10% of the dataset to train the linear combination model of Equation (1) and optimize the coefficients, $w_1, w_2, w_3, w_4,$ using Algorithm (2). We tested different training dataset sizes for both datasets and did not find noticeable differences for the value of the coefficients of Equation (1) by increasing the training dataset size beyond 10% of the bug reports. After optimizing the coefficients, we used the remaining 90% of the dataset for testing our online approach. With the arrival of each bug report in the test set, its stack trace is compared to stack traces of all previous bug reports in the testing set and then using the obtained coefficients from the training phase, the similarity of the bug reports is calculated. Next, the K nearest neighbor approach is used for predicting the severity of the incoming bug report. For instance, assume our test set has N bug reports, in our online severity prediction method, with the arrival of the *M'th* bug report (*0<M<=N*), all the *M-1* previous bug reports in the test set are compared to that bug report and their similarity is calculated using the linear model of Equation (1). The K nearest neighbor is then used to predict the severity of the *M'th* incoming bug report.

This approach aims to be practical in the sense that it can be deployed in an actual bug tracking system. This is because bug reports follow a temporal order based on their creation date. This also explains why we did not follow the traditional splitting of the data into 70% training and 30% testing sets, typically used in machine learning studies.

4.1.3 Dealing with Imbalanced data

The distribution of severities of bugs in both datasets is not balanced. Overall, there are more bugs having Critical or Major severities than other severity labels. The distribution of the severity labels in our datasets is shown in Figures 6 and Figure 7. Note that for Gnome, the severity label distribution is shown using the logarithmic scale since the dataset is much more imbalanced compared to Eclipse

These figures show that the distribution of severity labels in Eclipse and Gnome are unbalanced, favoring Critical and Major Severity labels. In the next section, we discuss the approach that is used to tackle the imbalanced dataset distribution problem.

4.2. Cost-sensitive learning

In an ideal scenario, the distribution of labels in the training set should be balanced (there are similar sample sizes for each label). Unfortunately, this scenario is not common for large industrial systems. For example, in Eclipse and Gnome datasets, some severities have less bug reports in the bug tracking system and the distribution of the labels is unbalanced.

Training a classifier on an imbalanced dataset creates bias towards the majority class labels. This is due to the fact that the classifier tends to increase the overall accuracy, which leads to ignoring minority class samples in the training set. Different approaches exist to overcome the imbalanced dataset problem. These approaches include oversampling the minority class, under-sampling the majority class or creating cost sensitive classifier [ZM03]. We experimented with all these approaches and observed that cost-sensitive learning [ZM03] is the most suitable approach to overcome the imbalanced dataset problem in Eclipse and Gnome datasets.

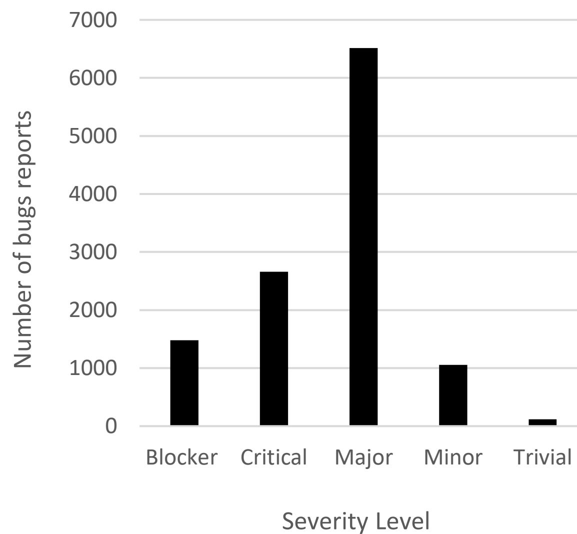

Figure 6. Distribution of the severity labels in Eclipse dataset

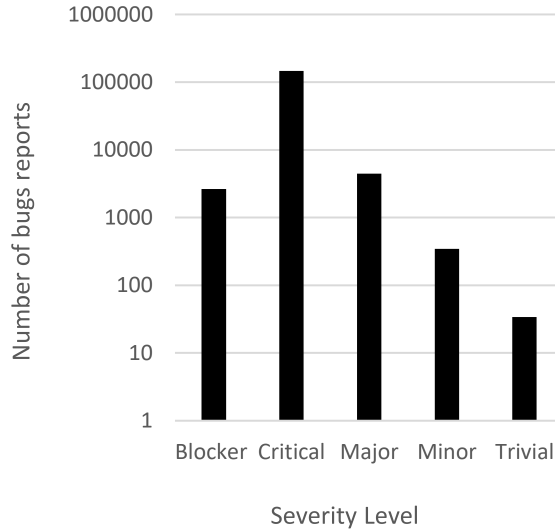

Figure 7. Distribution of the severity labels in Gnome dataset

To transform a classifier into a cost-sensitive classifier, we need the output of the classifier to be equal to the probability of a bug belonging to each severity class. Furthermore, we need to define a cost matrix. Using a cost matrix, the probability of each label is replaced by the average cost of choosing that class label. Indeed, to change a classifier to a cost-sensitive classifier, we do not need to change the internal functionality of the classifier. Instead, according to the output probabilities and using a cost matrix, the classifier makes an optimal cost-sensitive prediction.

In Equation (10), the K nearest neighbor returns the weight for each label, then the label with the largest weight is chosen. To make our classifier cost sensitive, we first need to adjust the outputs to represent probabilities instead of weights. For example, if *B* is a bug report in the testing dataset that has *m* classes. The classifier must provide a list of probabilities $p_1 \dots \dots \dots p_m$, in which each p_i shows the probability that the bug (*B*) severity label belongs to the ith class in the test set. Since, the summation of all probabilities should be equal to one (i.e., $p_1 + p_2 + \cdots + p_m = 1$), the weights need to be normalized.

To calculate the probability of each label, considering that the output of our classifier is $w_1 \dots \dots \dots w_m$, we use Equation (11) [SHH16], where W=$w_1 + \cdots + w_m$.

$$p_i = \frac{w_i}{W} \qquad (11)$$

The probability of each label is then changed to classification cost of each class label by 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 [LD13] in Figure 8.

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

Figure 8. Confusion matrix

In the confusion matrix, higher values of true positives and true negatives are favorable, thus we set the misclassification cost of these two values to zero. However, the cost of false positives and false negatives is selected based on the classification context.

We overcome the imbalance distribution problem by assigning high misclassification cost to the under-sampled class labels and low misclassification cost to over-sampled class labels. We chose the cost of the misclassification of each class label to be reciprocal to the number of existing instances of that class divided by the number of instances of the majority class (see Equation (12)). In this case, classes which have lower number of instances will have higher misclassification cost and classes which have high number of instances will have lower misclassification cost. If we consider having C different classes and assume 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 Equation (12) [SHH16].

$$MC_j = \frac{s}{s_j} \qquad (12)$$

The cost matrix is constructed using the misclassifications cost based on Equation (12). Then, we need to calculate the classification cost of each class label based on the cost matrix and the calculated probabilities based on Equation (11). 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 (13) [QWZZ13].

$$CCO_i = \sum_{j \in m \text{ and } j \neq i} CO_j \times P_j \quad (13)$$

Finally, the class label with the lowest classification cost is selected as the output of the classifier. Misclassification costs, if assigned improperly, may degrade the classification accuracy of the classifier. In this paper, we used Equation (12) to determine the misclassification costs. Furthermore, to avoid very high misclassification costs that may be associated to some class labels, we choose a threshold of ten to be the maximum misclassification cost.

We choose ten as our threshold because we want the majority and minority class labels to have the same impact on our classification method in both best- and worst-case scenarios based on Equation (13). In one extreme case, if all ten returned most similar bug reports are of the majority class label, then the classification cost of all other severity labels will be increased by 100. The reason is that based in Equation (12) the misclassification cost of the majority label is one and the probability of the majority class label is 100% if all the returned bug reports have majority class label. Then based on Equation (13), the classification cost of all other labels will be raised to 1*100=100. We also choose to set the threshold of misclassification cost to be ten because in the worst scenario if only one out of ten most similar bug reports has the minority class label (if we consider that all ten most similar bug reports have the same distance to the incoming bug report) then the misclassification cost of minority severity label is ten and the probability of minority class label is 10% too. In other words, based on Equation (13), the classification cost of all other severity labels will be increased by 10*10=100, which is the same as in the case for majority class labels. However, not in all the cases the distance of bug report to all ten nearest returned bug reports is the same and a better threshold could be identified by testing different values for threshold and comparing the results of the classification method.

Based on the proposed cost sensitive K nearest neighbor method, we update our testing phase to consider the probability of each severity and the cost of classifying each bug report to that severity using the misclassification cost matrix. Finally, the severity with the least classification cost will be selected as the output label (Figure 9 describes this process).

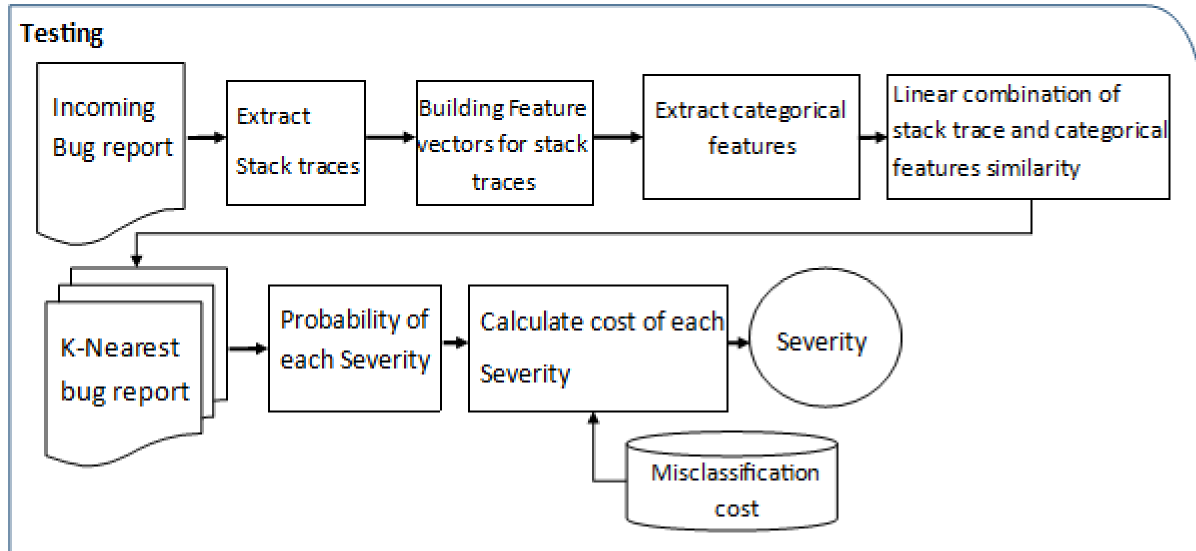

Figure 9. Updated the testing phase using cost sensitive k nearest neighbor

4.3. Predicting severity of bugs using bug report description

In this study, we compare our approach to the use of bug report descriptions. To predict the severity of a bug using the bug report description, we extract descriptions from all bug reports in our dataset. We tokenize words in the description of bug reports by splitting the text using space and new line character. We used the raw tokenized words for building a feature vector for each bug report. We build the feature vector using the distinct words in all bug report descriptions in our dataset. We use TF-IDF to weigh the feature vectors. In this approach, each bug report description is represented by a vector built based on the frequency of occurrence of each word (Equation (4)) multiplied by inverse document frequency of each word (Equation (5)). We follow the same online severity prediction approach that is discussed in Section 3 for predicting severity based on weighted feature vectors constructed from bug report descriptions.

We create an ordered set of bug reports based on their creation date. We then exercise the scenario in which each bug report in our ordered set is compared to all previous bug reports using their corresponding weighted feature vectors. After calculating the distance of each bug report to all the previous bug reports, we use the K nearest neighbor bug reports to determine the bug severity label using Equation (10). Based on the K nearest neighbor of Equation (10), the severity of an incoming bug report is selected based on the severity label of its nearest neighbor weighted using their distance to the incoming bug report.

Furthermore, we tackle the imbalanced dataset distribution problem by using the cost sensitive k nearest neighbor method with the same setting as for our approach using stack traces and categorical features. In this approach, after calculating the probability of each severity label, we use cost sensitive k nearest

neighbor of Equation (13) which calculates the cost of choosing each severity label by considering the misclassification cost of each severity label calculated using Equation (12) to predict severity of bug report. For example, assume our test set contains sorted bug reports $\{B_1, \dots \dots, B_N\}$, to predict severity of B_J (1 < J <= N), we compared B_J description to all the bug reports in the set $\{B_1, \dots, B_{J-1}\}$, then we predict the severity of B_J using Equation (13).

4.4. Predicting severity of bugs using a random classifier

We also compare the result of our approach to a random classifier model, which selects a severity label for each bug in proportion to the different class labels. Assume we have T severity classes and the number of bug reports that belong to each class is $B_{S_1} \dots \dots B_{S_T}$. If we randomly select severity labels for each bug, then our accuracy of predicting the severity label S_i can be calculated using Equation (14):

$$Accuracy\ (S_i) = \frac{B_{S_i}}{\sum_{j=1}^{T} B_{S_j}} \quad (14)$$

4.5. Evaluation Metrics

In this study, we use the precision, recall and F-measure metrics to evaluate our approach. If we consider the number of bugs for which we predict that they should have a severity label S_L as P_{S_L} and the number of bugs for which we correctly predict the severity label to be S_L as C_{S_L} then precision is defined by Equation (15):

$$Precision\ (S_L) = \frac{C_{S_l}}{P_{S_l}} \quad (15)$$

Furthermore, if we consider the number of bugs that actually have the severity label S_L as T_{S_L} then recall is calculated by Equation (16):

$$Recall\ (S_L) = \frac{C_{S_l}}{T_{S_l}} \quad (16)$$

In this study, a confusion matrix is built for each severity label. Then using the built confusion matrix, recall and precision is calculated. Since precision is the ratio of correctly predicted labels of a specific severity to the total number of labels predicted to have that severity, it actually measures the correctness of the approach. Meanwhile, since recall is the number of correct prediction of a severity to the total number of instances of that severity, it actually shows the completeness of the approach. However, the common practice is to combine these two metrics together to have a better perception of the accuracy of the severity prediction results. A common approach for combining these two metrics is F-measure. F-measure is the harmonic mean of precision and recall. F-measure is calculated according to Equation (17).

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \qquad (17)$$

4.6. Results and Discussion

In the rest of the paper we refer to the approach that uses stack traces alone as BSP_{ST}, the approach that uses bug report descriptions as BSP_{DE}, and the approach that uses stack traces and categorical features as BSP_{ST+CF}.

In this section, we discuss the results of applying the proposed approach to stack traces and categorical features of the bug reports of the Eclipse and Gnome datasets. In our previous study [SHH16], we showed that BSP_{ST} outperforms BSP_{DE}. In this study, we show how BSP_{ST+CF} performs better than BSP_{ST} and BSP_{DE}. When using a K nearest neighbor classifier, one of the most important factors is the value of K. The value of K shows the number of most similar items, which are used to choose a label. In our experiments, we recorded precision and recall and calculated F-measure by varying the value of K from 1 to 10 for each severity label in each dataset.

Figure 10 to Figure 19 show the results for predicting different severity levels by varying list size for the Eclipse and Gnome datasets. We revisit the research questions in light of the results presented in Figure 10 to Figure 19.

We provide the detailed values of precision, recall and F-measure for each of the list sizes for all severity levels for both datasets in Appendix A.

RQ1) F-measure of BSP_{ST}

Figure 10 to Figure 19 show the F-measure of the BSP_{ST} compared to the F-measure of BSP_{DE}. For Eclipse dataset severity prediction using BSP_{ST} outperforms BSP_{DE} for Critical and Blocker severity labels. BSP_{ST} has the same performance as BSP_{DE} for Major and Trivial severity labels. For Gnome dataset severity prediction using BSP_{ST} outperforms BSP_{DE} for Major and Blocker severity labels. BSP_{ST} has the same performance as BSP_{DE} for Critical and Trivial severity labels.

For both datasets, BSP_{DE} outperforms BSP_{ST} for the Minor severity label only. It is worth to mention that although the bug report descriptions contain stack traces, we have higher accuracy using stack traces independently as described in our approach. These results are consistent with our previous study [SHH16], confirming that BSP_{ST} outperforms or has the same performance BSP_{DE} for predicting all bug severity levels, except the Minor severity level, when applied to Eclipse and Gnome bug reports datasets.

We further investigated the reason that BSP_{ST} is slightly less performant compared to BSP_{DE} when predicting the Minor severity level. We elaborate more on this at the end of this Section.

We also compared the performance of our approach to a random classifier. Based on Figure 10 to Figure 19, we see that our approach outperforms a random classifier for all severity labels for both datasets, except for the Critical severity level in the case of the Gnome dataset. This is caused by two factors including the fact that the Critical severity label is the majority class label in Gnome and that the distribution of labels in Gnome favors the majority class. As we can see from Figure 7, the number of bugs of Critical severity is over 100,000, which is considerably higher than any other class.

Based on Figure 6, the number of bug reports in Eclipse with Major severity is excessively higher than other severity labels. Using a classifier, the excessive number of Major severity label instances creates a bias in the outcome of classifier by drifting the machine learning approach toward predicting a major class label to increase the overall accuracy. This explains the similar results in the severity prediction performance using BSP_{ST} and BSP_{DE} in Figure 12.

The accuracy of predicting the Trivial severity level using both approaches is low compared to other severity labels in both datasets. The reason is due to the fact that the number of bug reports having Trivial severity is considerably less than other severity labels.

RQ2) Sensitivity of the approach to the number of nearest neighbors

Based on the Figure 10 to Figure 19, the proposed approach is slightly sensitive to the number of nearest neighbors chosen. For Gnome, none of the approaches are hugely sensitive to the number of nearest neighbors. For Eclipse, only the Major severity is slightly sensitive to the number of neighbors. The reason is that the Major severity label based on the Figure 6 is the majority class label. In this case, increasing the number of nearest neighbors will cause more labels to appear in the returned list, which increases the probability of not choosing a Major severity label.

RQ3) Severity prediction improvement by adding categorical features

As shown in Figure 10 to Figure 14, for Eclipse dataset, we have the best performance using BSP_{ST+CF} compared to BSP_{ST} and BSP_{DE}. Also based on Figure 15 to Figure 19, we have the best performance using BSP_{ST+CF} compared to BSP_{ST} and BSP_{DE} for all cases but Minor severity for Gnome dataset. We can conclude that using the categorical features (product, component and operating system) in addition to stack traces improves the prediction accuracy of BSP_{ST}. The severity prediction accuracy improvement by adding categorical features ranges from 5% for Eclipse to 20% for the Gnome dataset.

Similar to RQ1, we found that our approach, BSP_{ST+CF}, performs better than a random classifier, except for the Critical severity level in the case of the Gnome dataset for the same reasons we explained in RQ1.

The results shown in Figure 10 to Figure 19 confirm that combining product, component, and operating system categorical features with stack traces increase the severity prediction accuracy.

We used the two tailed Mann-Whitney test to further assess the statistical significance of difference between the results shown in Figure 10 to Figure 19. More precisely, we compared F-measure of BSP_{ST+CF} to F-measure of BSP_{ST} and also F-measure of BSP_{ST} to F-measure of BSP_{DE}. We consider the difference between two sets of F-measures to be statistically significant if the significance level (p-value) is less than 0.05.

Based on the results shown in Appendix A, we found that, compared to BSP_{DE}, BSP_{ST} and BSP_{ST+CF} improve precision more than recall. Higher precision means that the predicted severity level using BSP_{ST} or BSP_{ST+CF} compared to BSP_{DE} has a higher probability of being the correct severity level. This makes the approach very suitable for developers who are reviewing bug reports, since it helps them to properly prioritize bugs and work on the critical ones sooner.

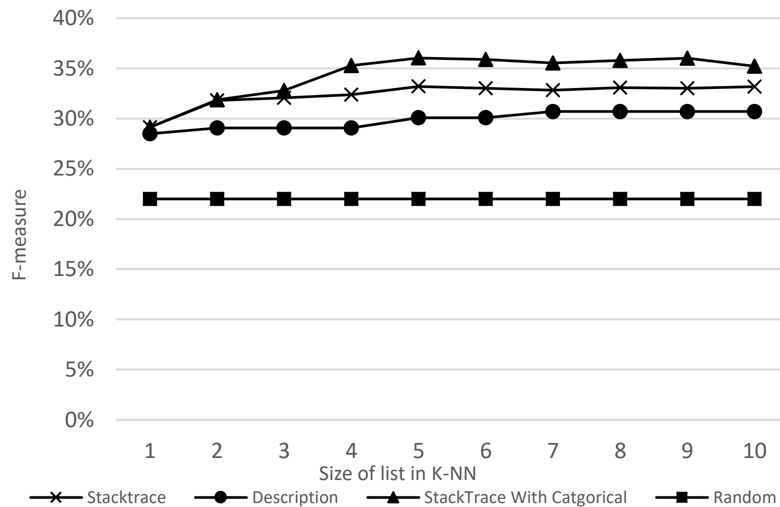

Figure 10. F-measure of predicting Critical severity levels by varying list size in Eclipse dataset

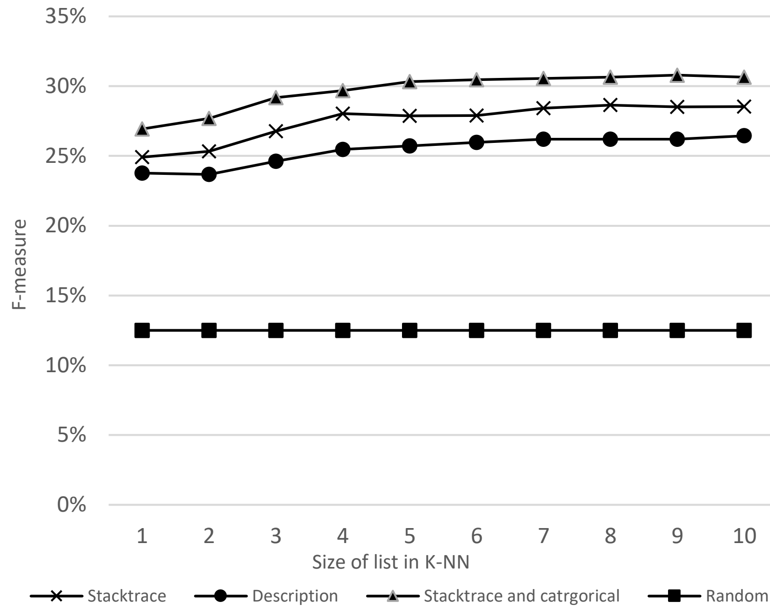

Figure 11. F-measure of predicting Blocker severity levels by varying list size in Eclipse dataset

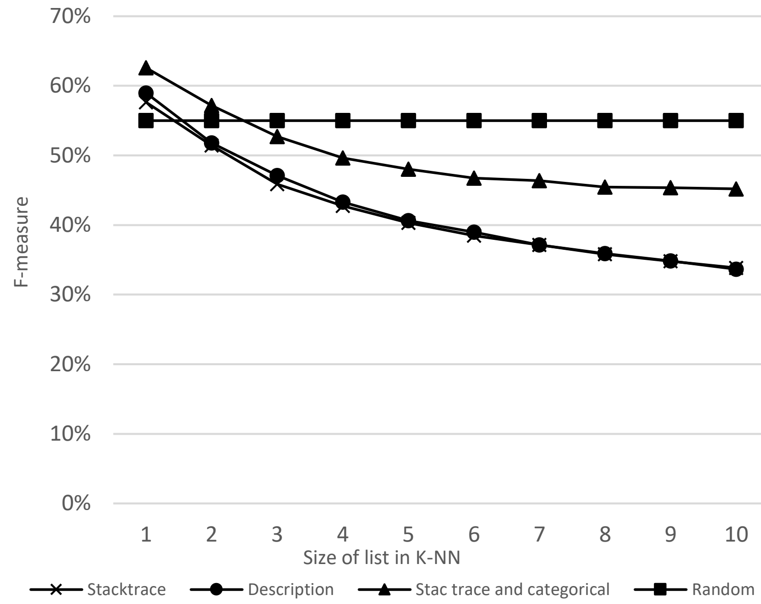

Figure 12. F-measure of predicting Majority severity levels by varying list size in Eclipse dataset

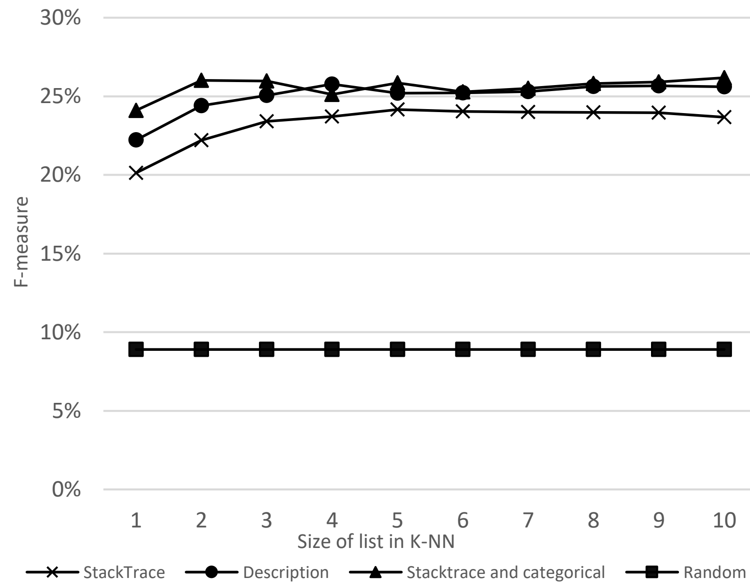

Figure 13. F-measure of predicting Minor severity levels by varying list size in Eclipse dataset

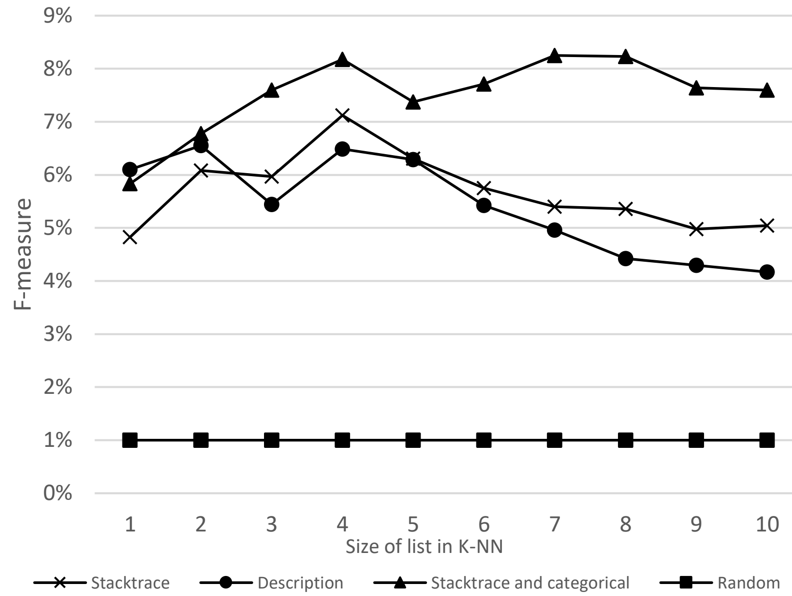

Figure 14. F-measure of predicting Trivial severity levels by varying list size in Eclipse dataset

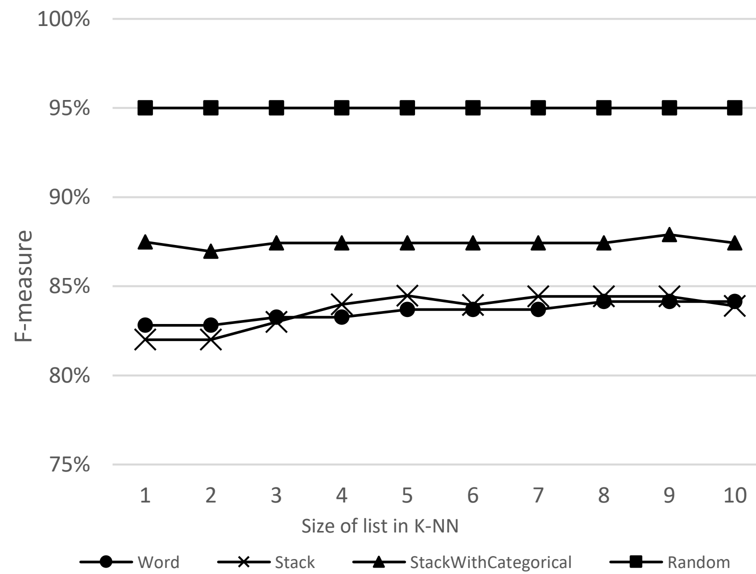

Figure 15. F-measure of predicting Critical severity levels by varying list size in the Gnome dataset

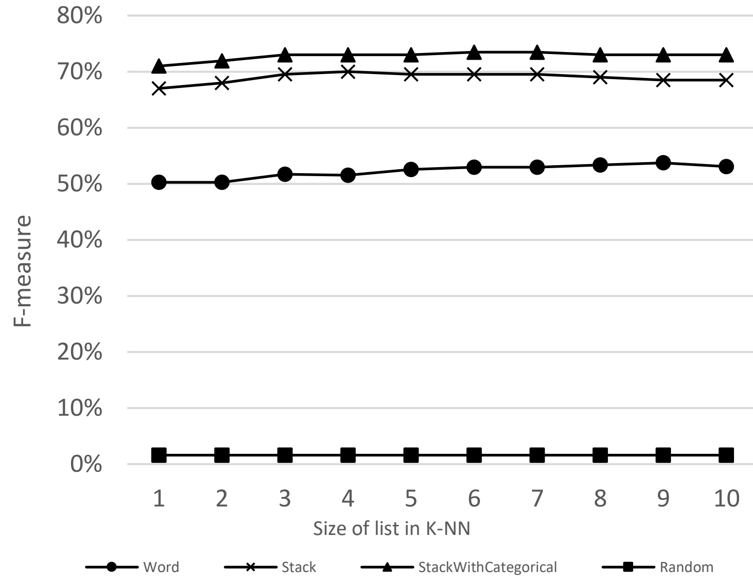

Figure 16. F-measure of predicting Blocker severity levels by varying list size in the Gnome dataset

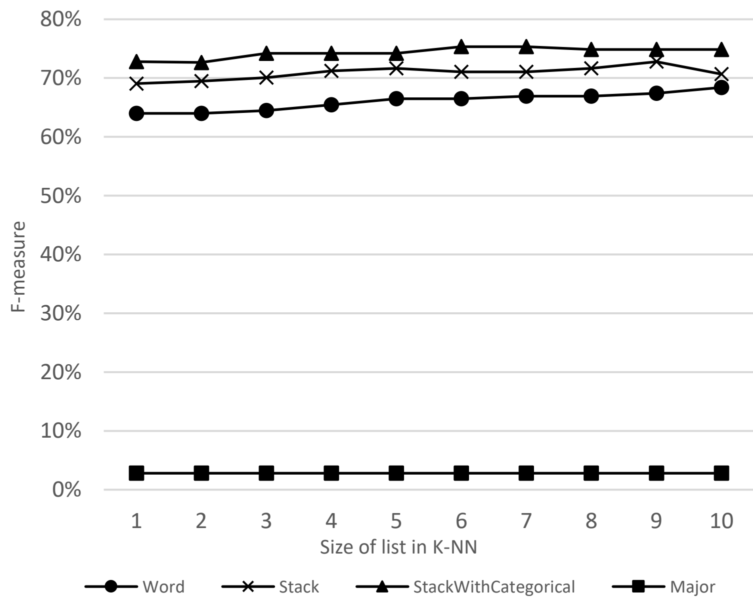

Figure 17. F-measure of predicting Major severity levels by varying list size in the Gnome dataset

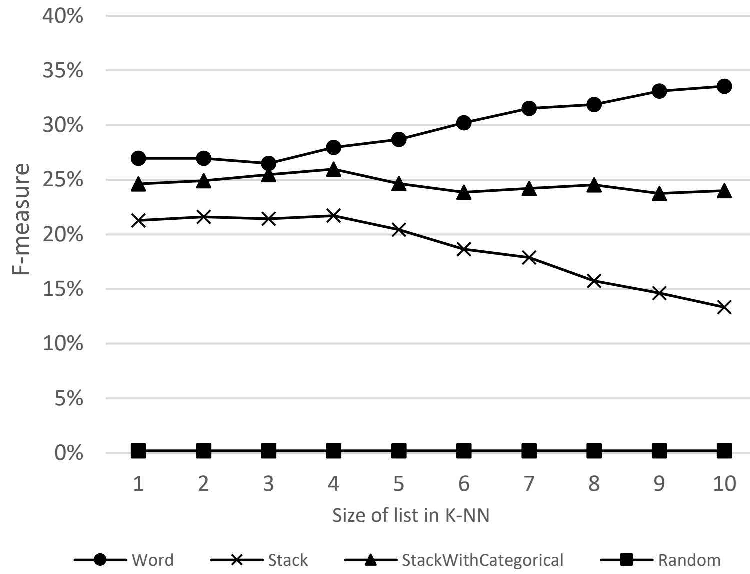

Figure 18. F-measure of predicting Minor severity levels by varying list size in the Gnome dataset

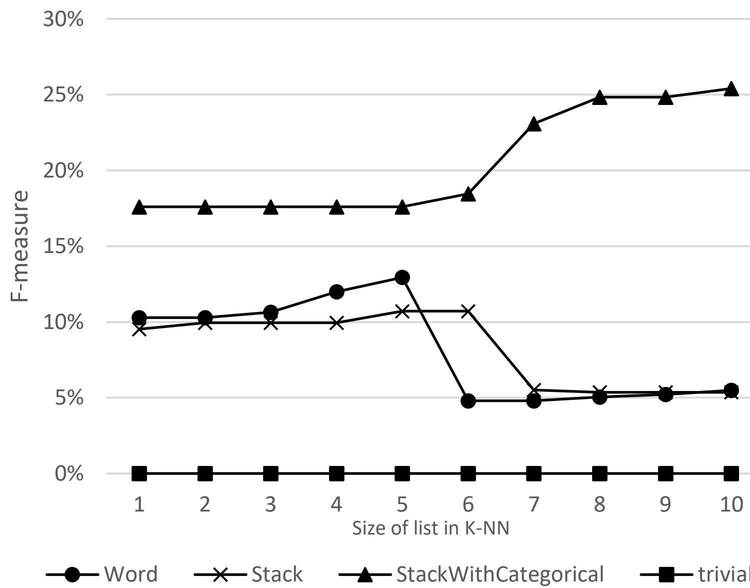

Figure 19. F-measure of predicting Trivial severity levels by varying list size in the Gnome dataset

Furthermore, we used the Cliff's non-parametric effect size measure, which shows the magnitude of effect size of difference between two sets of F-measures. The effect size estimates the probability that a value chosen from one group is statistically higher than a value chosen from another group [RJ05, GER11]. We want to calculate Cliff's effect size to estimate the probability that one F-measure obtained from one of our approaches is statistically higher than an F-measure of another approach. Cliff's effect size (*d*) is calculated as follows [RJ05, GER11]:

$$\text{Cliff's effect size} = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 * n_2} \quad (18)$$

In Equation (18), x_1 and x_2 are F-measure values within each group (each approach), # indicates number of times values of one group is higher or lower than values of other group and $n_1$and n_2 are size of each approach result. More precisely, let us define d_{ij} as follows [RJ05, GER11]:

$$d_{ij} = \begin{cases} +1 \; if \; F - measure_i \; from \; first \; approach \; > \; F - measure_j \; from \; second \; approach \\ -1 \; if \; F - measure_i \; from \; first \; approach \; < \; F - measure_j \; from \; second \; approach \\ 0 \; if \; F - measure_i \; from \; first \; approach = \; F - measure_j \; from \; second \; approach \end{cases} \quad (19)$$

Based on Equation (19), we can define Cliff's effect size as follows [RJ05, GER11]:

$$\text{Cliff's effect size} = \frac{\sum_i \sum_j d_{ij}}{n_1 * n_2} \quad (20)$$

Cliff's effect size ranges from [-1, +1]. Cliff's effect size of +1 indicates that all the values of the first group are larger than second group, and -1 shows that all values of the first group are smaller than the second group. Considering Cliff's effect size as *d*, effect size is small when $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$ [RJ05, GER11]:

Table 2. Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and categorical features and an approach which uses stack traces alone using the Eclipse dataset

	Critical	Blocker	Major	Minor	Trivial
Two tailed Mann-Whitney test significance level	0.045	0.007	0.02	0.0001	0.0007
Cliff's effect size	0.59	0.72	0.62	0.98	0.90

Table 3. Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and categorical features and an approach which uses stack traces alone using the Gnome dataset

	Critical	Blocker	Major	Minor	Trivial
Two tailed Mann-Whitney test significance level	0.0001	0.00018	0.00028	0.00018	0.00018
Cliff's effect size	1	1	0.97	1	1

Tables 2 and 3 show the result of Mann-Whitney test significance level and Cliff's effect size test between BSP_{ST+CF} and BSP_{ST} for Eclipse and Gnome, respectively. For both datasets and all severity levels, the difference of F-measure of the two approaches is statistically significant with a p-value < 0.05. Hence, we can conclude that the difference between F-measures of BSP_{ST+CF} and $BSP_{ST}$is statistically significant for both datasets.

Furthermore, since for all severity labels, the Cliff's effect size of BSP_{ST+CF} compared to BSP_{ST} is more than 0.474, we can conclude that the former outperforms the latter with a large effect size for both datasets.

Table 4. Mann-Whitney test significance level and Cliff's effect size of the approach with stack traces and an approach that uses bug report descriptions using the Eclipse dataset

	Critical	Blocker	Major	Minor	Trivial
Mann-Whitney test a significance level	0.001	0.00736	0.79486	0.00168	0.72786
Cliff's effect size	0.88	0.72	-0.08	-0.84	0.1

Table 4 and Table 5 show the result of Mann-Whitney test significance level and Cliff's effect size between BSP_{ST} and BSP_{DE} for Eclipse and Gnome respectively.

For Eclipse dataset, the difference of F-measure of the two approaches is statistically significant with a p-value < 0.05 for all severity levels, except for Major and Trivial. Consistent to the result of our previous study [SHH16] as shown in the Figure 12, since the Major severity is the majority class label, and the classifier is normally biased toward majority class label, we have the same accuracy when using BSP_{ST} as when using BSP_{DE}. For Trivial, as shown in Figure 14, since the Trivial severity level is under-sampled, we see the same effect. This is because the classifier is biased toward the majority class label.

Table 5. Mann-Whitney test significance level and Cliff's effect size of the approach that uses stack traces and an approach that uses bug report descriptions using the Gnome dataset

	Critical	Blocker	Major	Minor	Trivial
Mann-Whitney test a significance level	0.34722	0.00018	0.00018	0.00018	0.8493
Cliff's effect size	0.26	1	1	-1	0.06

Furthermore, for Critical and Blocker, the Cliff's effect size of BSP_{ST} compared to BSP_{DE} is more than 0.474 and BSP_{ST} outperforms BSP_{DE} with a large effect. For the Major severity level, the Mann-Whitney test and Cliff's effect size test results are consistent. Consistent with Mann-Whitney test, Cliff's effect size for Major severity is close to zero which shows that the difference in F-measure of two approaches is not statistically different. Also, for the Minor severity level, the Mann-Whitney test shows that the difference between F-measure of the two approaches is statistically significant and the Cliff's effect size shows that BSP_{DE} largely outperforms BSP_{ST}.

For the Gnome dataset, the difference of F-measure of the two approaches is statistically significant with a p-value < 0.05 for all severity levels, except for Critical and Trivial. This is similar to the Eclipse dataset.

Furthermore, for all severity labels other than Critical and Trivial, the Cliff's effect size is large. For Blocker and Major severity, Cliff's effects size of BSP_{ST} compared to BSP_{DE} is more than 0.474. For the Critical severity level, consistent with Mann-Whitney test which shows that the difference of F-measure of the two approaches is not statistically significant, Cliff's test shows the effect size is small. Also, for Trivial severity level, we have consistent results from Mann-Whitney test and Cliff's test (Mann-Whitney p-value>0.05 and Cliff's test is close to zero) which both show the difference of F-measure of BSP_{ST} and BSP_{DE} are not statistically significant and both approach have the same performance. For the Minor severity level, the Mann-Whitney test shows that the difference of F-measure values of the two approaches is statistically significant and the Cliff's effect size shows BSP_{DE} outperforms BSP_{ST} by a large effect size.

For the Eclipse dataset, in the case of Minor severity, we had lower performance when using BSP_{ST} compared to using the BSP_{DE}. We investigated the dataset to study the underlying reason. Based on this investigation, we arrived to the following possible reasons:

- Bug reports from Eclipse with Minor severity are mainly not from Eclipse platform, but from Eclipse plugins. Hence, their stack traces significantly vary from one to another. Studies [LM13, SHL17] show that BSP_{ST} can detect duplicate bug reports with higher accuracy compared to the approach which uses bug report descriptions. In our previous study [SHH16], we confirmed this

argument and showed that BSP_{ST} can predict bug reports severities with higher accuracy than BSP_{DE}. However, in the case of bug reports with Minor severity, due to the variety among plugins, there are fewer duplicate bug reports. Fewer duplicate bug reports mean less similar stack traces and hence a lower severity prediction accuracy.

- We studied bug reports with Minor severity and observed that many bug reports with Minor severity have categorical features available in their header or description. Since categorical information is stored in the header or description of the bug reports, when using BSP_{DE}, categorical features are being used to predict severity too. These categorical features boost severity prediction accuracy of bug report descriptions. For example, Bug Report#296383 and Bug Report#313534 of Eclipse bug repository are shown in Table 6 and Table 7 respectively.

Table 6. Eclipse Bug Report #296383

Bug report field	Value
Product	EclipseLink
Component	JPA
Header	***JPA*** 2.0 server test script requires better rebuild integration with Eclipse IDE development
Description	If you are developing in eclipse while running these server tests - note that there is currently an issue with the ***JPA*** 2.0 test framework where a full build in eclipse will remove classes expected by the server test ant script. *You will see the following issue unless you do a full ant trunk build after any Eclipse IDE rebuild or clean .Do a 2nd rebuild off of trunk or ***eclipselink***.jpa.test after any IDE development.

In both bug reports, JPA, the faulty component of the bug, appeared in the header of the bug reports. This extra information provides a ground to make predictions based on BSP_{DE} outperform predictions based on BSP_{ST}. However, as shown in Figure 13, if we add categorical features to stack traces, then the BSP_{ST+CF} outperforms the one that uses only BSP_{DE}.

For the Gnome dataset (Figure 15 to Figure 19), the only case that the BSP_{ST+CF} is outperformed by the BSP_{DE} is when the approach is predicting Minor severity. We investigated the reason and observed that the description of bugs with Minor severity contain more structured information than categorical features of the bugs in Gnome Bugzilla. For example, as shown in Table 8, Gnome Bug Report#273727, which is a bug

with Minor severity, not only contains all important categorical features in its description, but also it contains information regarding the step to reproduce the bug. Moreover, in addition to categorical features, sometimes the source code is copied in the description of the bug report. For example, Gnome Bug Report# 532680, shown in Table 9, contains the source code of the bug. This information causes the approach that uses descriptions to outperform the approach which uses stack traces and categorical features when predicting the Minor severity.

Table 7. Eclipse Bug Report #313534

Bug report field	Value
Product	EclipseLink
Component	JPA
Header	***JPA***: entities in separate eclipse project must be explicitly listed in persistence.xml
Description	<exclude-unlisted-classes> has no effect in this configuration Configuration: (I am using an eclipse .classpath reference only) - and not generating a jar file - no MANIFEST.MF Class-Path entry - no <jarfile> element in persistence.xml - persistence.xml is in client project - no persistence.xml in entities project .classpath = <classpathentry combineaccessrules="false" kind="src" path="/org.eclipse.persistence.example.***jpa***.server.entities"/> <classpathentry kind="src" path="src"/> This may be expected behavior for SE PU's when we fail to use the persistence.xml element - as mormally the entities must be at the root of the classpath that contains persistence.xml <jarfile>entities.jar</jarfile>>Found: Using <exclude-unlisted-classes>false</exclude-unlisted-classes><class>org.eclipse.persistence.example.***jpa***.server.business.Cell</class> [EL Config]: 2010-05-19 10:24:01.195--ServerSession(27196165)--Thread(Thread[main,5,main])--The access type for the persistent class [class org.eclipse.persistence.example.***jpa***.server.business.Cell] is set to [FIELD].

Table 8. Gnome Bug Report#273727

Bug report field	Value
Product	evolution
Component	Shell
Header	sanity check for e-d-s on start
Description	Descriptionorogor 2005-03-15 19:41:19 UTC Distribution: Gentoo Base System version 1.4.16 Package: Evolution Priority: Normal Version: GNOME2.8.1 2.0.3 Gnome-Distributor: Gentoo Linux Synopsis: Random UI crash Bugzilla-Product: Evolution Bugzilla-Component: Calendar Bugzilla-Version: 2.0.3 BugBuddy-GnomeVersion: 2.0 (2.8.1) Description: Description of the crash: It just exit Steps to reproduce the crash: 1. Do random stuff , preferably around the contact list 2. wait 3. play with it again 4 Expected Results: than it doesn t exit How often does this happen? maybe average session is 15 minutes (depends how mutch you use the ui)

Table 9. Gnome Bug Report #532680

<table>
<tr><th>Bug report field</th><th>Value</th></tr>
<tr><td>Product</td><td>GIMP</td></tr>
<tr><td>Component</td><td>Plugins</td></tr>
<tr><td>Header</td><td>help-browser segfaults on 64bit systems</td></tr>
<tr><td>Description</td><td>It was the latest Ubuntu HH version of the libgtk2.0-0 package.
% dpkg -p libgtk2.0-0
Package: libgtk2.0-0
Architecture: amd64
Source: gtk+2.0
Version: 2.12.9-3ubuntu3
Now it is immediately clear what went wrong:
_gdk_x11_convert_to_format has parameter src_buf==NULL
This shows that we have a pixmap that is non-NULL, with pixmap->pixels that is NULL. This pixmap is found in the cache. If I insert the condition
&& gdk_pixbuf_get_pixels(icon->pixbuf) != NULL

in gtk+-2.12.9/gtk/gtkiconfactory.c around line 2445:

@@ -2441,7 +2441,8 @@
if (icon->style == style &&
icon->direction == direction &&
icon->state == state &&
- (size == (GtkIconSize)-1 || icon->size == size))
+ (size == (GtkIconSize)-1 || icon->size == size) &&
+ gdk_pixbuf_get_pixels(icon->pixbuf) != NULL)
{
if (prev)
{
then the segfault goes away. I have not investigated further what the real cause of the problems is.</td></tr>
</table>

5. Threats to Validity

In this Section, we explain threats to the *external* validity, *internal* validity and *construct* validity of our approach.

5.1 Threats to External Validity

We evaluated our approach using two well-known open source datasets. While the results of our experiments show that leveraging categorical features improves the severity prediction accuracy, in order to generalize these results, our approach needs to be tested on a bigger pool of datasets.

Moreover, for Eclipse, stack traces are stored in the description of the bug reports and are optional. Only 10% of Eclipse bug reports contain stack traces. This said, since 2015, Eclipse has been equipped with an automated stack trace collection system. We intend to extend our study to include bug reports beyond 2015 in a future study.

We evaluated our approach using the severity labels that are reported from users and further revised and adjusted (if need be) by the triagers. Errors may occur when reporting or adjusting severity levels, which can impact our analysis.

5.2 Threats to Internal Validity

One of the sources of internal threats is the misclassification function used in our approach. In our study we used Equation (12) to calculate the misclassification cost. This misclassification cost equation was derived based on heuristics. While the results obtained using Equation (12) are convincing, using a more optimized approach for deriving misclassification cost of each severity label could further improve the severity prediction accuracy.

We set a threshold of ten to be our upper bound for the misclassification cost based on the criteria we explained in Section 4.2. A different threshold may have yield different results.

The regular expression used for extracting stack traces from bug report descriptions may have missed some stack traces or some functions in the stack traces. This could reduce the accuracy of the severity prediction approach. Using a different regular expression may improve the results.

6. Related Work

There exist two categories of severity prediction techniques based on the features that are used.

- The first category uses descriptions of bug reports. They resort to natural language processing techniques to calculate similarity among bug report descriptions and predict the bug severity.

- The second category uses stack traces. They consider methods in stack traces as terms in description and calculate similarity of bug reports based on similarity of their stack traces.

6.1 Predicting Bug Severity Using Description

Antoniol et al. [AADPKG08] built a data set using 1800 issues reported to bug tracking systems of Mozilla, Eclipse and JBoss (600 reports from each bug tracking system). In these bug tracking systems, issues can be labeled as corrective maintenance or other kind of activities such as perfective maintenance, preventive maintenance, restructuring or feature addition. In this study issues that are related to corrective maintenance are categorized as *Bug*, while issues related to other activities are categorized as *Non-bug*. Each of 1800 issues, extracted from bug tracking systems, are revised and labeled manually. In some cases, bugs were not related to Eclipse, so they are removed from the training and testing set. They considered bug reports descriptions as the best sources of information to train the machine learning techniques for predicting severity [BJSWPZ08]. The authors used words in the descriptions as features. They used frequency of words for weighing the feature vector which corresponds to each bug. In addition to the words in the description, they added the value of the severity field as a feature. After extracting words, they are stemmed. However, no stop-word removal is performed. The rationale behind not removing stop-words is that they may be discriminative and may be used by the classifier to improve classification accuracy. They used various classification methods, such as decision trees, logistic regression, and naïve Bayes to classify issues. Each of the classifiers is trained using the top 20 or 50 features. The accuracy of the approach when applied to Mozilla is 67% and 77% with top 20 and 50 features respectively. The accuracy of the approach applied to Eclipse issues having 20 features is 81% and having 50 features is 82%. The accuracy of the approach when applied to JBoss issues having 20 features is 80% and having 50 features is 82%. They showed that for Eclipse, some words (e.g. “Enhancement”) are good indicator of Non-bug issues, while some words (e.g. “failure”) are good indicator of Bug issues. They also showed that their approach outperforms regular expression-based approach which uses grep command [AADPKG08].

Menzies et al. [MM08] did a study on an industrial system in NASA. NASA uses a bug tracking system called *Project and Issue tracking system* (PITS). They examined bugs raised by testers and sent to PITS. In NASA severity of bugs are in a five-point scale. One corresponds to the worst, most critical bug and five is the dullest bug. They introduced a tool called SEVERIS which, using the description of the bugs and text mining techniques, predicts the severity of an issue. They tokenized terms, removed stop-words and finally stemmed the terms. They used TF-IDF score of each term to rank them, then they cut all but top K features. Furthermore, they did another round of feature reduction using information gain. They used rule learner to deduce rules from the weighed features. For the case study, five different systems and consequently five

different datasets are used. The main problem with the datasets was that they did not have any bug with severity one (Critical severity) and the total number of bugs was 3877. The authors calculated precision, recall and F-measure for each of the severities. Using top 100 words as features, F-measure was averagely 50% for predicting bugs severities. The most important point in this study is that they showed, in their dataset, using top 3 features or 100 features does not change the F-measure significantly. This fact shows that predicting severities using much less number of features that have more discriminative power reveals good results.

Lamkanfi et al. [LDGG10] did a study to show discriminability power of the terms in description. They build dataset of the open source software such as Mozilla, Eclipse and Gnome to evaluate the proposed approach. In Bugzilla, severity can be Critical, Major, Normal, Minor, Trivial or Enhancement. To have a coarse-grain categorization the authors labeled all issues which were Critical or Major as Severe and issues that were Minor, Trivial or Enhancement as Non-Severe. They ignored using issues marked as Normal because it is the default choice which will be assigned to a bug in Bugzilla and users may choose it arbitrarily. The bug severity prediction in this study is modeled as a document classification problem. They used the summary and description of bug reports for evaluation. When extracting bugs, the authors organized them according to the products and components that are affected. They used 70% of bugs as training set and 30% of issues as testing set. They did a study on the most important features and concluded that words like “crash” or “memory” are good indicators of severe bugs. The authors did a second round of experiments using only descriptions and showed that using only descriptions decreases the performance in many cases. They did experiment having training sets with different sizes and concluded that a training set having 500 bugs is enough to have a generalizable result. They also showed that increasing the number of bugs to more than 500 does not change the evaluation result. In the fourth round of experiments, they showed that applying severity prediction approach on Eclipse bug tracking system isolating bugs according to the affected component and product leads to a better result.

Lamkan et al. [LDSV11] compared the effect of having diverse mining algorithms applied to bug repositories to predict the severity of the bugs. The authors used the same labeling principal as Lamkanfi et al. [LDGG10]. Eclipse and Gnome are the datasets that are used to evaluate the accuracy of predicted severities. Bug reports are extracted and categorized according to their faulty products and components. They compared Naive Bayes, Naive Bayes multinomial, 1-Nearest Neighbor and Support Vector machine classifiers. Due to the fact that different classification approaches need different ways to weigh feature vector, in this study when doing experiments using Naïve Bayes, only presence or absence of each term is used for weighing features. When doing experiments using Naïve Bayes Multinomial, frequency of each

word is used for weighing each feature vector. Using 1-Nearest Neighbor or Support Vector Machine, term frequency and inverse document frequency is used for weighing feature vectors. They calculated precision and recall, the area under curve (AUC) for each dataset to compare the classifiers. They showed that using Naïve Byes Multinomial, the area under curve is averagely 80% which is higher than other approaches. Next, they showed that using Naïve Bayes classifier, a stable accuracy value is achieved having 250 bug reports of each severity for training. This shows that increasing training set by adding more than 250 bug reports does not change the accuracy. Having a list of words that have good discriminative power, they concluded that each component has its own list of words. Thus, terms that have good discriminative power are component specific. This result encourages applying severity prediction approaches on each component independently since it reveals better results.

Yang et al. [YHKC12] compared effectiveness of feature selection methods on a coarse grain severity prediction technique. They used Naïve Bayes classifier to study effectiveness of each feature selection method. They used information gain, Chi-square and correlation coefficient as feature selection techniques. They used Eclipse and Firefox datasets and true positive rate (TPR), false positive rate (FPR) and area under curve (AUC) metrics to evaluate their studies. They showed high information gain is a good indicator of severe bugs and low information gain is a good indicator of non-severe bugs. They concluded that the best feature selection technique for Eclipse and Mozilla is the correlation coefficient.

Tian et al. [TLS12] did a study on a finer grain severity prediction approach. Features with more discriminability power is a necessity for a fine grain severity prediction. They used pruning techniques to improve the discriminability power of features. For each of the bugs in the training set they used unigram and bi-gram techniques to extract features from textual description. These features are then used to calculate the similarity of descriptions of bug reports in the testing and training set. To calculate similarity of bug reports descriptions, they used an extended version of BM25 called $BM25_{ext}$. The similarity of bug report descriptions using unigram and bi-gram features together with the similarity of categorical features including faulty products and components are then used to retrieve most similar bugs in the training set. They used a linear combination of these four features to calculate similarity of bug reports. The K nearest neighbor method is then used to classify the bug reports in the testing set. The rational for using K nearest neighbor is that the most similar bug reports are expected to have the same severity. They used Open Office, Mozilla and Eclipse to evaluate their approach. They showed that their approach outperforms SEVERIS.

Yang et al. [YCKY14] studied the effectiveness of four quality indicators of bug reports in severity prediction. They used a naïve Bayes classifier and did a coarse grain severity prediction. The four quality

indicators studied in their work are stack traces, report length, attachment and steps to reproduce. They did two series of experiments on 2 components of Eclipse to measure the effectiveness of quality indicators on predicting severity of bug. In the first series of experiment they only studied the impact of the existence of those quality indicators. In the second round of experiments they used the quantitative value of those indicators. They extracted these quantitative values from each quality indicators differently. For stack traces they used number of functions, for attachment they considered number of attachments, for report length they used the quantitative value of report length and for step to reproduce they used number of steps. They showed that stack trace is the most useful information which could be used as an indicator. However, unlike Sabor et al. [SHH16], they did not study the effectiveness of using the content of stack traces for predicting severity of bugs.

Yang et al. [YZL14] studied the effectiveness of topic modeling on fine grain severity prediction. Instead of using categorical features in similarity calculation, they only considered bugs if they had same product, component and priority. In the first step, they used latent dirichlet allocation (LDA) to extract topics from the corpus of documents. They represented each topic as a bag of words. Instead of vector space model they used smoothed unigram vectors and they used KL divergence instead of cosine similarity to measure similarity of smoothed vectors. They showed effectiveness of their approach by applying it on Mozilla, Eclipse and NetBeans bug repositories. They used an online approach in which with the incoming of each bug report its probability vector is extracted. Next, K nearest neighbor is applied to the bugs with the same topic and according to returned list of similar bugs, the label is chosen.

Bhattachrya et al. [BINF12] explored alternate avenues for bug severity prediction by a graph-based analysis of the software system. They built graph based on different aspects of software systems including source code graphs based on function calls (i.e., a static call graph) modules (module collaboration graph) and, in a more abstract level, they built a graph based on developer's collaboration. They used Firefox, Eclipse and MySql to show effectiveness of their approach. They used various graph-based metrics including average degree, clustering coefficient, node rank, graph diameter and assortativity to characterize software structure and evolution. They showed that node rank metric in the function call graph is a good indicator of bug severity. They also showed that that Modularity Ratio is a good indicator for modules which need less maintenance effort.

Zhang et al. [ZYLC15] explored the effectiveness of concept profile in predicting severity of bugs. They extracted concept terms and calculated their threshold for each fine grain severity label in the training set. A concept profile corresponding to each severity label using concept terms is built next. Instead of vector space model, they used probability vector to represent each bug. Also, instead of cosine similarity, they

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. They showed effectiveness of their approach by applying it on Eclipse and Mozilla bug repositories. They used an offline approach in which they used 90% of the data as the training set and 10% of the data as the testing set. They showed that their approach outperforms other machine learning approaches.

6.2 Predicting Bug Severity Using Stack Traces

There exist studies which uses advanced machine learning methods based on stack traces to predict various bug report fields [SNTH18]. Sabor et al. [SHH16] used the content of stack trace for predicting severity of bugs. They extracted the stack traces form the Eclipse bug reports. Next, they built a feature vector based on the functions of all stack traces in the bug repository. They weighed the feature vector using term frequency and inverse document frequency. They used cosine similarity to compare feature vectors. In their online approach, with the incoming of each bug report the feature vector is built and the K nearest neighbor is returned. Since not all of the bug reports had the same distance to the incoming bug report in the reported list, they used distance-based K nearest neighbor [GDZX12]. Since the distribution of labels was not balance, they used a cost sensitive K nearest neighbor [QWZZ13] to predict severity of the incoming bug report. They applied their approach on the Eclipse bug repository for the period of 2001 to 2015 and showed that their approach using stack traces outperforms the approach that uses bug report descriptions.

Most existing approaches rely on bug report descriptions for predicting bug report severity. These approaches are presented in Section 6.1. In our previous work, we showed that BSP_{ST} could predict severity of bugs with a better accuracy compared to BSP_{DE}. In this paper, we showed that adding categorical features can further enhance the severity prediction accuracy.

7. Conclusions and Future Work

In this paper, we proposed a new severity prediction approach. The new approach combines the use of stack traces and categorical features. Our approach uses a linear combination of stack trace similarity and categorical features similarity to predict the severity of bugs. Since the dataset (i.e., the labels in bug report repositories) is normally unbalanced, we adopted a cost sensitive K nearest neighbor approach to overcome the unbalanced dataset distribution problem.

We used two open source and popular datasets (Eclipse and Gnome bug repositories) to evaluate our approach. The result showed that in both cases the accuracy of predicting the severity of bugs is higher using BSP_{ST} compared to BSP_{DE}. Moreover, adding categorical features and using linear combination of

stack trace and categorical features similarity can further improve the severity prediction accuracy for both datasets.

In this study we used K nearest neighbor in an online approach to predict severity of bugs, in the future we want to assess the effect of using more advanced machine learning methods such as Support Vector Machine (SVM), Random Forest and Neural Networks. We believe using more advanced machine learning techniques may further improve the severity prediction accuracy. Moreover, we plan to extend our study by applying our approach to a wider range of public and proprietary bug repository datasets. Also, we plan to use a more optimized misclassification cost function to overcome the unbalance dataset distribution problem.

APPENDIX A

Detailed precision and recall

In Section 4.7, we showed the F-measure of approaches that predict severity of bugs using stack traces and categorical features, using stack traces only, and using bug report descriptions. We present the detailed precision, recall and F-measure values for all the k value (list size of the k nearest neighbor) here.

Table A1. Severity prediction accuracy (Eclipse Critical severity)

List size	Bug report description			Stack traces			Stack trace and categorical features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	28.2%	29.1%	28.64%	31.5%	27.1%	29.1%	34.5%	25.1%	29.14%
2	26.4%	33.1%	29.37%	30.6%	33.1%	31.8%	33.9%	30.1%	31.88%
3	26.3%	33.1%	29.31%	30%	34.4%	32%	34.3%	31.4%	32.785
4	26.3%	33.1%	29.31%	30.3%	34.8%	32.4%	33.3%	37.5%	35.27%
5	27.5%	34%	30.4%	31.1%	35.6%	33.2%	34.1%	38.2%	36.03%
6	27.2%	34.1%	30.26%	31.1%	35.2%	33	33.3%	38.9%	35.88%
7	28%	34%	30.7%	31.3%	34.5%	32.8%	33.3%	38.1%	35.53%
8	28%	34%	30.7%	31.6%	34.7%	33.1%	33.3%	38.7%	35.79%
9	28%	34%	30.7%	31.6%	34.6%	33%	34%	38.3%	36.02%
10	28%	34%	30.7%	31.7%	34.8%	33.2%	32.6%	38.3%	35.22%

Table A2. Severity prediction accuracy (Eclipse Blocker severity)

List size	Bug report description			Stack trace			Stack trace and categorical features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	23%	24.6%	23.77%	26.5%	23.5%	24.91%	27.7%	26.2%	26.92%
2	20%	29.4%	23.8%	23.2%	27.9%	25.33%	24.5%	31.8%	27.67%
3	20%	32%	26.76%	23%	32%	26.76%	24.2%	36.7%	29.16%
4	20%	35.3%	28%	23.2%	35.4%	28.03%	23.8%	39.4%	29.67%
5	20%	36.1%	27.87%	22.7%	36.1%	27.87%	24.2%	40.6%	30.32%
6	20%	37.3%	27.89%	22.5%	36.7%	27.89%	24.1%	41.4%	30.46%
7	20%	38%	28.42%	22.7%	38%	28.42%	24%	42%	30.54%
8	20%	38%	28.63%	22.8%	38.5%	28.63%	23.8%	43%	30.64%
9	20%	38.1%	28.5%	22.6%	38.6%	28.50%	23.8%	43.6%	30.79%
10	20%	39%	28.53%	22.5%	39%	28.53%	23.5%	44%	30.63%

Table A3. Severity prediction accuracy (Eclipse Major severity)

List size	Bug report description			Stack trace			Stack trace and categorical features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	60%	58%	62.59%	61.7%	54.1%	57.65%	63.1%	62.1%	62.59%
2	61.1%	45%	57.16%	63.5%	43.2%	51.41%	64.7%	51.2%	57.16%
3	62%	38%	52.69%	64.5%	35.6%	45.87%	65.9%	43.9%	52.69%
4	63.3%	33%	49.63%	65.5%	31.7%	42.72%	66.5%	39.6%	49.63%
5	63%	30.1%	48.03%	66%	29%	40.29%	66.8%	37.5%	48.03%
6	64.1%	28%	46.76%	66.3%	27.1%	38.47%	67.4%	35.8%	46.76%
7	65%	26%	46.35%	66.2%	25.8%	37.12%	67.5%	35.3%	46.35%
8	65%	24.8%	45.44%	66.5%	24.5%	35.80%	67.7%	34.2%	45.44%
9	65%	23.8%	45.37%	66.9%	23.5%	34.78%	68.6%	33.9%	45.37%
10	65%	22.7%	45.19%	66.5%	22.7%	33.84%	68.2%	33.8%	45.19%

Table A4. Severity prediction accuracy (Eclipse Minor severity)

List size	Bug Report Description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	23.1%	21.4%	22.21%	23.5%	17.6%	20.12%	28.1%	21.1%	24.10%
2	20.9%	29.3%	24.39%	20.8%	23.8%	22.19%	24.6%	27.6%	26.01%
3	19.6%	34.7%	25.05%	19.8%	28.6%	23.4%	22.1%	31.5%	25.97%
4	19.2%	39.2%	25.77%	18.8%	32.1%	23.71%	20.1%	33.5%	25.12%
5	18.2%	40.9%	25.19%	18.5%	34.8%	24.15%	19.9%	36.9%	25.85%
6	17.8%	43.35	25.22%	18%	36.2%	24.04%	19.1%	37.4%	25.28%
7	17.5%	45.7%	25.30%	17.6%	37.7%	23.99%	18.9%	39.2%	25.50%
8	17.6%	47.1%	25.62%	17.3%	39%	23.96%	18.7%	41.6%	25.80%
9	17.5%	48.2%	25.67%	17.1%	40%	23.95%	18.5%	43.2%	25.90%
10	17.3%	49.2%	25.59%	16.8%	40.1%	23.67%	18.6%	44.2%	26.18%

Table A5. Severity prediction accuracy (Eclipse Trivial severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	6.7%	5.6%	6.10%	5.5%	4.3%	4.82%	7%	5%	5.83%
2	5.5%	8.1%	6.55%	5.5%	6.8%	6.08%	6.4%	7.2%	6.77%
3	4.1%	8.1%	5.44%	4.6%	8.5%	5.96%	6.2%	9.8%	7.59%
4	4.6%	11%	6.48%	5.1%	11.8%	7.12%	5.9%	13.3%	8.17%
5	4.4%	11%	6.28%	4.3%	11.8%	6.30%	5.1%	13.3%	7.375
6	3.6%	11%	5.42%	3.8%	11.8%	5.74%	5.2%	14.9%	7.70%
7	3.2%	11%	4.95%	3.5%	11.8%	5.39%	5.5%	16.5%	8.25%
8	2.8%	10.5%	4.42%	3.4%	12.6%	5.35%	5.4%	17.3%	8.23%
9	2.7%	10.5%	4.29%	3.1%	12.6%	4.975	4.9%	17.3%	7.63%
10	2.6%	10.5%	4.16%	3.1%	13.5%	5.04%	4.8%	18.2%	7.6%

Table A6. Severity prediction accuracy (Gnome Critical severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	**79%**	**87%**	**82.80%**	**82%**	**82%**	**82%**	**86%**	**89%**	**87.47%**
2	**79%**	**87%**	**82.80%**	**82%**	**82%**	**82%**	**85%**	**89%**	**86.95%**
3	**79%**	**88%**	**83.25%**	**83%**	**83%**	**83%**	**85%**	**90%**	**87.42%**
4	**79%**	**88%**	**83.25%**	**83%**	**85%**	**83.98%**	**85%**	**90%**	**87.42%**
5	**79%**	**89%**	**83.70%**	**83%**	**86%**	**84.47%**	**85%**	**90%**	**87.42%**
6	**79%**	**89%**	**83.70%**	**82%**	**86%**	**83.95%**	**85%**	**90%**	**87.42%**
7	**79%**	**89%**	**83.70%**	**82%**	**87%**	**84.42%**	**85%**	**90%**	**87.42%**
8	**79%**	**90%**	**84.14%**	**82%**	**87%**	**84.42%**	**85%**	**90%**	**87.42%**
9	**79%**	**90%**	**84.14%**	**82%**	**87%**	**84.42%**	**85%**	**91%**	**87.89%**
10	**79%**	**90%**	**84.14%**	**81%**	**87%**	**83.89%**	**85%**	**90%**	**87.42%**

Table A7. Severity prediction accuracy (Gnome Blocker severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	54%	47%	50.25%	67%	67%	67%	70%	72%	70.98%
2	54%	47%	50.25%	67%	69%	67.98%	70%	74%	71.94%
3	56%	48%	51.69%	69%	70%	69.49%	72%	74%	72.98%
4	57%	47%	51.51%	70%	70%	70%	72%	74%	72.98%
5	58%	48%	52.52%	70%	69%	69.49%	72%	74%	72.98%
6	59%	48%	52.93%	70%	69%	69.49%	72%	75%	73.46%
7	59%	48%	52.93%	70%	69%	69.49%	72%	75%	73.46%
8	60%	48%	53.33%	69%	69%	69%	735	73%	73%
9	61%	48%	53.72%	70%	67%	68.46%	73%	73%	73%
10	61%	47%	53.09%	70%	67%	68.46%	73%	73%	73%

Table A8. Severity prediction accuracy (Gnome Major severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	64%	64%	64%	75%	64%	69.06%	77%	69%	72.78%
2	64%	64%	64%	76%	64%	69.48%	78%	68%	72.65%
3	64%	65%	64.49%	76%	65%	70.07%	79%	70%	74.22%
4	64%	67%	65.46%	76%	67%	71.21%	79%	70%	74.22%
5	65%	68%	66.46%	77%	67%	71.65%	79%	70%	74.22%
6	65%	68%	66.46%	77%	66%	71.07%	79%	72%	75.33%
7	65%	69%	66.94%	77%	66%	71.07%	79%	72%	75.33%
8	65%	69%	66.94%	77%	67%	71.65%	78%	72%	74.88%
9	65%	7%	67.40%	77%	69%	72.78%	78%	72%	74.88%
10	66%	71%	68.40%	76%	66%	70.64%	78%	72%	74.88%

Table A9. Severity prediction accuracy (Gnome Minor severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	26%	28%	26.96%	26%	18%	21.27%	32%	20%	24.61%
2	26%	28%	26.96%	27%	18%	21.6%	33%	20%	24.90%
3	26%	27%	26.49%	295	17%	21.43%	35%	20%	25.45%
4	29%	27%	27.96%	30%	17%	21.70%	37%	20%	25.96%
5	32%	26%	28.68%	32%	15%	20.42%	39%	18%	24.63%
6	36%	265	51.62%	33%	13%	18.65%	40%	17%	23.85%
7	40%	26%	31.51%	35%	12%	17.87%	42%	17%	24.20%
8	44%	25%	31.88%	37%	10%	15.74%	44%	17%	24.52%
9	49%	25%	33.10%	39%	9%	14.62%	46%	16%	23.745
10	51%	25%	33.55%	40%	8%	13.33%	48%	16%	24%

Table A10. Severity prediction accuracy (Gnome Trivial severity)

List size	Bug report description			Stack trace			Stack trace and categorical Features		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
1	12%	9%	10.28%	23%	6%	9.51%	33%	12%	17.6%
2	12%	9%	10.28%	29%	6%	9.94%	33%	12%	17.6%
3	13%	9%	10.63%	29%	6%	9.94%	33%	12%	17.6%
4	18%	9%	12%	29%	6%	9.94%	33%	12%	17.6%
5	23%	9%	12.935	5%	6%	10.71%	33%	12%	17.6%
6	12%	3%	4.8%	5%	6%	10.71%	40%	12%	18.46%
7	12%	3%	4.8%	34%	3%	5.51%	50%	15%	23.07%
8	16%	3%	5%	25%	3%	5.35%	72%	15%	24.82%
9	20%	3%	5%	25%	3%	5.35%	72%	15%	24.82%
10	33%	3%	5.5%	25%	3%	5.35%	83%	15%	25.40%

References

[AADPKG08] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Guéhéneuc, “Is it a bug or an enhancement?: a ext-based approach to classify change requests,” *Proc. of the IBM International Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON)*, 2008.

[BINF12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, “Graph-based analysis and prediction for software evolution,” *Proc. of the 34th International Conference on Software Engineering (ICSE)*, pp. 419-429, 2012.

[BJSWPZ08] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?” *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE),* pp. 308-318, 2008.

[DMJ11] J. L. Davidson, N. Mohan, and C. Jensen, “Coping with duplicate bug reports in free/open source software projects,” *Proc. of IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC)*, pp. 101–108, 2011.

[ETIHK19] N. Ebrahmi, A. Trabeslsi, Md. S. Islam, A. Hamou-Lhadj, K. Khanmohammadi, "An HMM-Based Approach for Automatic Detection and Classification of Duplicate Bug Reports," Elsevier Journal of Information and Software Technology (IST), Volume 113, pp. 98-109 , 2019.

[EH15] N. Ebrahimi, A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata," *Proc. of the IBM 25th Annual International Conference on Computer Science and Software Engineering (CASCON),* pp. 201-210, 2015.

[EIHH16] N. Ebrahimi, Md. S. Islam, A. Hamou-Lhadj, M. Hamdaqa, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," *Proc. of the IBM 26th Annual International Conference on Computer Science and Software Engineering (CASCON),* pp. 75-84, 2016.

[GDZX12] J. Gou, L. Du, Y. Zhang, T. Xiong, “A New Distance-weighted K nearest Neighbor Classifier,” *Journal of Information & Computational Science*, pp. 1429-1436, 2012.

[GER11] G. Macbeth, E. Razumiejczyk, R. Ledesma, "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations," Universitas Psychologica. vol.10, n.2, pp.545-555, 2011.

[LD13] A. Lamkanfi and S. Demeyer, "Predicting reassignments of bug reports an exploratory investigation," *Proc. of the 17th European Conference on Software Maintenance and Reengineering*, pp. 327–330, 2013.

[LDGG10] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," *Proc. of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 1–10, 2010.

[LDSV11] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," *Proc. of 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 249–258, 2011.

[LM13] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," *Proc. of European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 69-78, 2013.

[MHNSL15] 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," *Proc. of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 342-351, 2015.

[MM08] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," *Proc. of the International Conference on Software Maintenance (ICSM)*, pp. 346–355, 2008.

[MRS08] C. D. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA, 2008.

[NHTL16] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, A. Larsson, "A Bug Reproduction Approach Based on Directed Model Checking and Crash Traces," Wiley Journal of Software: Evolution and Process (JSPE), Volume 29, Issue 3, 2016.

[NHTL15] M. Nayrolles, A. Hamou-Lhadj, S. Tahar and A. Larsson, "JCHARMING: A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," *Proc. of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 101-110, 2015.

[PF13] F. Provost, T. Fawcett. Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking, O'Reilly Media, 2013.

[QWZZ13] Z. Qin, A. T. Wang, C. Zhang, and S. Zhang, "Cost-Sensitive Classification with k Nearest Neighbors," *Proc. of the 6th International Conference on Knowledge Science, Engineering and Management (KSEM),* pp. 112–131, 2013.

[RJ05] R.Grissom and J. Kim. Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates, 2005.

[SHH16] K. K Sabor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic prediction of the severity of bugs using stack traces," *Proc. of the 26th Annual International Conference on Computer Science and Software Engineering* (CASCON '16), pp. 96-105, 2016.

[SHL17] K. K. Sabor, A. Hamou-Lhadj and A. Larsson, "DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports," *Proc. of IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 240-250, 2017.

[SNTH18] K. K. Sabor, M. Nayrolles, A. Trabelsi and A. Hamou-Lhadj, "An Approach for Predicting Bug Report Fields Using a Neural Network Learning Model," *Proc. of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 232-236, 2018.

[SSG15] G. Sharma, S. Sharma, S. Gujral, "A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms," *Proc. of the 4th International Conference on Eco-friendly Computing and Communication Systems,* Volume 70, pp. 632-639, 2015.

[SLKJ11] C. Sun, D. Lo, S. C. Khoo and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," *Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 253-262, 2011.

[TLS12] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," Proc. of the 19th *International Working Conference on Reverse Engineering (WCRE)*, pp. 215-224, 2012.

[WFH11] I. H. Witten, E. Frank, M. A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, 2011, ISBN 9780123748560.

[YCKY14] C. Z. Yang, K. Y. Chen, W. C. Kao, and C. C. Yang, "Improving severity prediction on software bug reports using quality indicators," *Proc. of the 5th IEEE International Conference on Software Engineering and Service Science (ICSESS*), pp. 216–219, 2014.

[YHKC12] 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," *Proc. of the 19th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 240–249, 2012.

[YZL14] 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," *Proc. of the 38th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 97–106, 2014.

[ZCYLL16] T. Zhang, J. Chen, G. Yang, B. Lee, X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Elsevier Journal of Systems and Software*, Volume 117, pp. 166-184, 2016.

[ZYLC15] T. Zhang, G. Yang, B. Lee, and A. T. S. Chan, "Predicting severity of bug report by mining bug repository with concept profile," *Proc. of the 30th Annual ACM Symposium on Applied Computing (SAC)*, pp. 1553–1558, 2015.

[ZM03] J. Zhang, I. Mani, "KNN Approach to Unbalanced Data Distributions: A Case Study," *Proc. of the ICML'2003 Workshop on Learning from Imbalanced Datasets, 2003.*