Information and Software Technology xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automatic prediction of the severity of bugs using stack traces and categorical features

Korosh Koochekian Sabor[a,*], Mohammad Hamdaqa[b], Abdelwahab Hamou-Lhadj[a]

[a] Software Behaviour Analysis (SBA) Research Lab, Department of Electrical and Computer Engineering, Concordia University, Montréal, QC, Canada
[b] School of Computer Science, Reykjavik University, Iceland

ARTICLE INFO

Keywords:
Bug severity
Stack traces
Machine learning mining software repositories
Software maintenance

ABSTRACT

Context: The severity of a bug is often used as an indicator of how a bug negatively affects system functionality. It is used by developers to prioritize bugs which need to be fixed. The problem is that, for various reasons, bug submitters often enter the incorrect severity level, delaying the bug resolution process. Techniques that can automatically predict the severity of a bug can significantly reduce the bug triaging overhead. In our previous work, we showed that the accuracy of description-based severity prediction techniques could be significantly improved by using stack traces as a source of information.

Objective: In this study, we expand our previous work by exploring the effect of using categorical features, in addition to stack traces, to predict the severity of bugs. These categorical features include faulty product, faulty component, and operating system. We experimented with other features and observed that they do not improve the severity prediction accuracy. 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. We adopted 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 Eclipse submitted between 2001 and 2015 and Gnome submitted between 1999 and 2015 show that the accuracy of our severity prediction approach can be improved from 5% to 20% by considering categorical features, in addition to stack traces.

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

1. Introduction

Defects in a software system that go undetected during the verification phase 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 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). This information is then used by developers to fix the bug.

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 is a tedious and time-consuming task. This is due to the large number of reports received [4]. 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 [10,26].

Despite the existence of guidelines on how to determine the severity level of a bug, studies show that the severity level is often assigned by the users incorrectly [12,28]. This is mainly due to the fact that users are not expert in the system domain. An inaccurate severity level causes

* Corresponding author.
E-mail addresses: k_kooche@ece.concordia.ca (K.K. Sabor), mhamdaqa@ru.is (M. Hamdaqa), abdelw@ece.concordia.ca (A. Hamou-Lhadj).

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

https://doi.org/10.1016/j.infsof.2019.106205
Received 15 June 2018; Received in revised form 25 July 2019; Accepted 17 October 2019
Available online xxx
0950-5849/© 2019 Published by Elsevier B.V.

delay in processing of bug reports as shown by Zhang et al. [28]. Triagers often have to review bug reports in order to determine the right severity level. This is usually done manually, hindering productivity and slowing down the bug resolution process [23,26,27,29].

To address this issue, several methods [1,10,13] have been developed 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 classify the incoming ones. They mainly use the words in the bug report description as the main features for classification. In our previous work [18], we showed that using the information in stack traces, which are attached to bug reports, severity of a bug could be predicted more accurately compared to using bug report description.

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 the development time [13]. For this reason, in this work, we extend our previous work by proposing a severity prediction approach that uses the combination the stack traces and the categorical features of bug reports, namely the product, the component, and the 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 respectively. We show that by combining stack traces and categorical features, the prediction accuracy of our previous approach that uses solely the 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 the bug 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 the bug description, the bug stack trace and other categorical features. In this Section, we explain the features that compose a bug report.

2.1. Bug report description

Bug report description is used to explain the defective behavior of the system. 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 usually verbose. An example of a well written description is "Download fails on Customer Report when user clicks on the downed button" meanwhile an example of an inefficient 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.

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 Fig. 1. The stack trace in Fig. 1 is taken from 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.

Studies have shown that stack trace is considered to be one the most useful information needed by developers in order to fix a bug [3]. 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 further help developers to fix the bug. Categorical features of a bug include the product, component, version, operating system, severity of the bug, submitters, etc. Each bug report categorical field (e.g. product, component, operating system and version) is assessed by a triaging team who may decide to update these field 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 severed. It is therefore natural 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 Witten et al. [24]. 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 do not improve accuracy. For this reason, we selected product, component, and operating system as our categorical features set.

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:

- Blocker[4] severity shows bugs that halt the development process. Blocking bugs are the bugs that do not have any work around.
- Critical1 bugs are bugs that cause loss of data or sever memory leaks.
- Major1 bugs are the bugs that are seriously obstacle to work with the software system.
- Normal1 bugs are advised to be chosen when the user is not sure about the bug or if the bug is related to documentation.
- Minor1 bugs are the ones which are worth reporting but does not interfere with the functionality of program.
- Trivial1 bugs are cosmetic bugs such as typo in the java docs.
- Enhancement1 bugs are the gray areas including new features that are not considered as bug.

3. The proposed approach

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

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

```
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()
```

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

then chosen based on the severity of the K nearest neighbor (K nearest bug report) to the incoming bug.

3.1. Predicting bug severity

In this paper, we follow an approach based on linear combination of stack traces and categorical features similarity to calculate similarity of bug reports and predict the severity of an incoming bug report.

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

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

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 Eq. (1), the similarity of two bug reports is the linear combination of their corresponding stack traces and categorical features similarities. In Eq. (1) similarity of categorical features is one if they are the same and zero if they are not. The method used to extract features (stack trace and categorical) will be explained in Section 3.2 and the method used for calculating similarity of stack traces will be explained in Section 3.3. The introduced similarity function in Eq. (1) has four tuneable parameters w_i: (w_1, w_2, w_3, w_4), which represent the weights of each feature. These four parameters must be optimized to reflect the importance of each feature. To tune these parameters, we used an adaptive learning approach that uses a cost function and gradient descent in a similar way to the one used by Sun et al. [21]. The format of our training set is similar to the one used by Sun et al. [22]. 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 which 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. 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. Since $S_1 = \{\mathrm{bug}_1, \mathrm{bug}_2\}$, $S_2 = \{\mathrm{bug}_3, \mathrm{bug}_4\}$, then we can create four triples with the format (*q, rel, irr*) for training: $(\mathrm{bug}_1, \mathrm{bug}_2, \mathrm{bug}_3)$, $(\mathrm{bug}_2, \mathrm{bug}_1, \mathrm{bug}_4)$, $(\mathrm{bug}_3, \mathrm{bug}_4, \mathrm{bug}_1)$, $(\mathrm{bug}_4, \mathrm{bug}_3, \mathrm{bug}_2)$.

As explained earlier, we need to define a cost function based on the created triple format to optimize the free parameters (w_1, w_2, w_3, w_4). The selected cost function RankNet [22] can be defined as follows

$$Y = Sim(irr, q) - Sim(rel, q) \tag{2}$$

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

Our goal is to minimize the defined RankNet cost function Eqn 3. The cost function is minimized when the similarity of bug reports with the same severity defined in Eq. (1) is maximized and the similarity of the bug reports with different severities is minimized. To minimize the above cost function, we used a gradient descent algorithm as shown in Algorithm 2.

3.2. Bug features extraction (stack trace extraction and categorical features extractions)

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 Fig. 2 [11]. Moreover, to extract stack traces form the Gnome bug repository, we defined and implemented the regular expression presented in Fig. 3.

Normally, categorical features can be found in the xml preview of a bug report. Each categorical features is enclosed by a different 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

Algorithm 1 Creating training set.

```
TS = ∅: training set
N>0 parameter controlling size of TS
G={$G_1, G_2, G_3, G_4, \dots, G_n$ }
$G_i$ = 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 2 Optimization using gradient descent [22].

```
TS = ∅: training set
N>0 size of TS
η: 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
      $x = x - \eta * \frac{\partial \mathrm{RankNet}}{\partial x}$
  End for
End for
End For
```

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

Fig. 2. Regular expression used for extracting stack traces from bug report description in the Eclipse bug repository [11].

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

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

same format. In our work, we implemented a custom parser to extract this information from the bug reports xml previews.

3.3. Stack trace similarity

We create feature vector from all distinct functions in all the stack traces, then we weigh the feature vector for each stack trace and compare the weighed feature vectors together using cosine similarity to compare stack traces together. We further elaborate on each of these steps below.

3.3.1. Building feature vectors

This Section explains how to map a stack trace to a feature vector, which is weighed by term-frequency and inverse document frequency (TF-IDF). In our approach the term concept refers to a function name and a document refers to a stack trace. However, before explaining the process let us consider the following definitions:

- Let $T = f_{1,} f_{2,} \ldots, f_L$ 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 $m = |\Sigma|$ that represents distinct function names (terms) in the system.
- Let $\Gamma = T_{1,} T_{2,} \ldots, T_K$ 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 Eq. (4):

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

In Eq. (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 Eq. (5):

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

In Eq. (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 $T_{i,} \in \Gamma$ 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. Similarity analysis of stack traces

To compare two stack traces, we measure the distance between their corresponding feature vectors using the cosine similarity measure. Given two vectors $V_1 = \langle v_{11}, v_{12}, \ \ldots\ v_{1n}\rangle$ and $V_2 =< v_{21}, v_{22}, \ \ldots\ v_{2n}>$, the cosine similarity is calculated as follows [14]:

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

As shown in Eq. (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 similarity of bug reports by a linear combination of their stack traces and categorical features similarity, we use KNN to predict bug report severity. KNN is an instance-based lazy learning algorithm [22]. 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 Eq. (7) [15].

$$C(X) = \underset{c_j\ \in C}{argmax} \qquad score\left(c_j,\ neighbors_k(X)\right) \tag{7}$$

In Eq. (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 Eq. (8) [15].

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

In Eq. (8), $class(y) = c_j$ returns either one or zero. It is one when $class(y) = c_j$ and zero otherwise. In Eq. (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 Eq. (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 Eq. (9), as discussed by Gou et al. [5], where $dist_i$ is the distance of bug

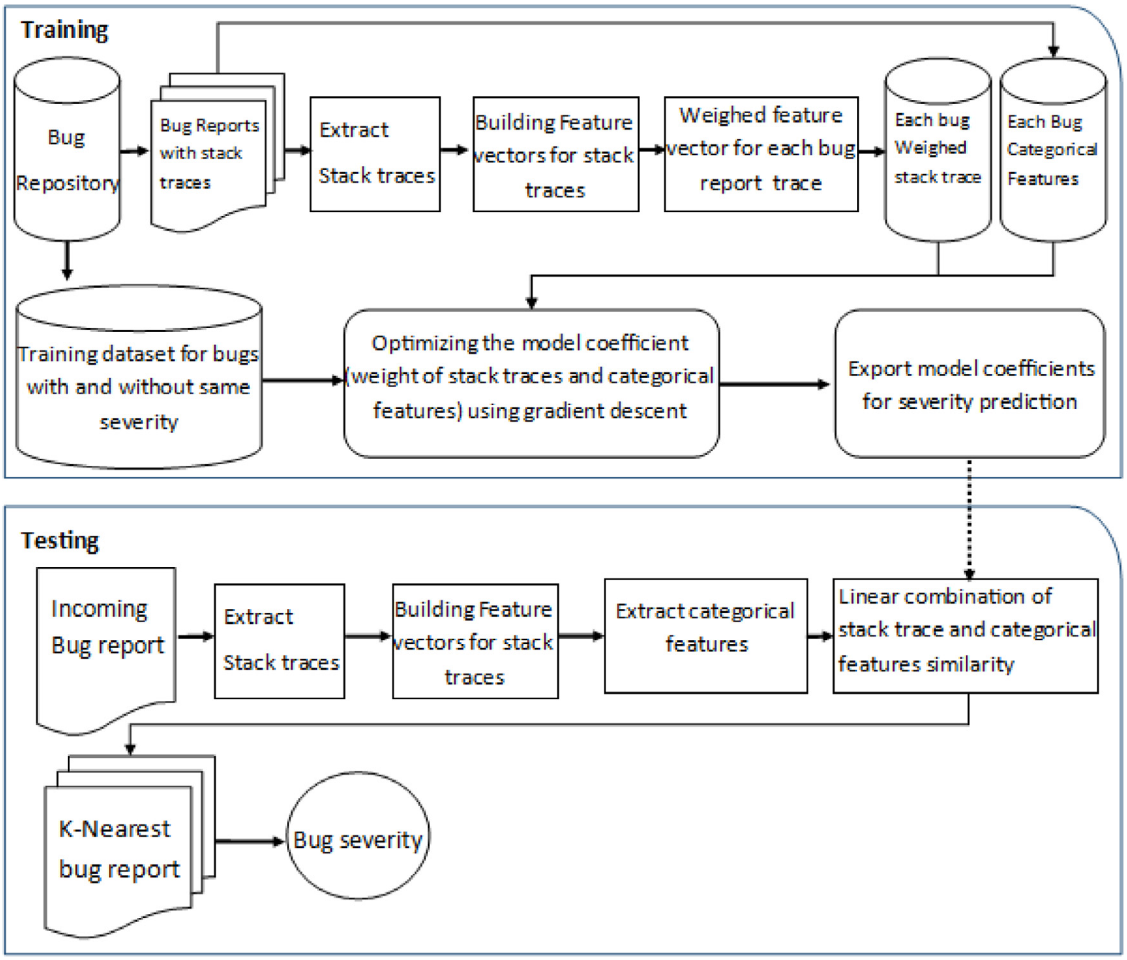
Training
Bug Repository
Bug Reports with stack traces
Extract Stack traces
Building Feature vectors for stack traces
Weighed feature vector for each bug report trace
Each bug Weighed stack trace
Each Bug Categorical Features
Training dataset for bugs with and without same severity
Optimizing the model coefficient (weight of stack traces and categorical features) using gradient descent
Export model coefficients for severity prediction
Testing
Incoming Bug report
Extract Stack traces
Building Feature vectors for stack traces
Extract categorical features
Linear combination of stack trace and categorical features similarity
K-Nearest bug report
Bug severity

Fig. 4. Overall approach.

report i.

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

Eq. (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 Eq. (8) to incorporate the weights calculated in Eq. (9). The updated score function is shown in Eq. (10) [15]:

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

where *w(x,y)* is the weight of each instance in the top *k* similar returned instances which is calculated by Eq. (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 Eq. (10), the label with the highest weight is selected as the output label.

3.5. Overall approach

We explain the Training and Testing phase of our online severity prediction method in this Section. Fig. 4 shows the overall approach.

3.5.1. Training

Our training phase is shown in Fig. 4. Training phase starts by extracting the main features (Section 3.2) that can be used to predict a bug severity; namely, the bug stack traces and the bug categorical features. Then, stack traces are mapped to weighed feature vectors. The details of how to create these feature vectors for stack traces is explained in Section 3.3.1. After creating the feature vectors for stack traces, similarity analysis (Section 3.3.2) are used to measure the similarity between stack traces of different bug reports. We use linear combination (Section 3.1) of the corresponding stack traces and categorical features similarity to take categorical features in consideration in the final similarity results. The linear model is trained using the training dataset build according to Algorithm 1 and optimized using the gradient descent (Algorithm 2). The tuned coefficients (w_1, w_2, w_3, w_4) are then used the testing phase.

3.5.2. Testing

Our online severity prediction (depicted in Fig. 4) works as follows: when the system receives a new bug report, the bug stack trace is extracted using the same regular expression used in the training part. Next, feature vector is built using all the distinct functions in the stack trace of new bug report and stack traces of all bug reports previous to that in the bug tracking system. The feature vector is weighed using TF-IDF, After that, the similarity between the weighed feature vector of the new bug report and the weighed feature vectors of the previous bug reports in the dataset (i.e., adjacency matrix representing all stack traces of previous bug reports) is calculated. The output of this part is a matrix, which shows how similar is the incoming bug-report's stack-trace to the stack traces of all the previous bug reports. Similarity of the incoming bug

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 dataset	1000	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

stack trace is then linearly combined with the similarity of its categorical features to all the previous bug reports using the tuned coefficients (w_1, w_2, w_3, w_4) from the training phase. Finally, the linear combined similarity is used to predict the severity of the incoming bug report using K nearest neighbor method.

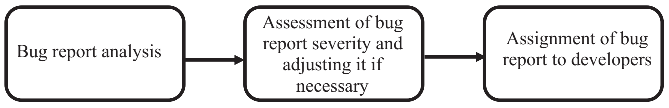
Bug report analysis
Assessment of bug report severity and adjusting it if necessary
Assignment of bug report to developers

Fig. 5. The bug handling process.

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 approaches using stack traces and the approach which uses bug reports descriptions only. 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 random approach?
RQ2. Using the KNN classifier for predicting bug severity, how different number of neighbors can affect the F-measure of the proposed approach?
RQ3. How adding categorical features to the stack traces affects the accuracy of the severity prediction compared to solely using stack traces information and the random approach?

Answering question 1 shows the importance of using stack traces as an alternative of using bug report descriptions. Answering question 2 shows the sensitivity of the proposed approach based on the number of nearest neighbors chosen. Answering question 3 explains how adding categorical features can contribute to predicting the severity of bugs.

4.1. Experimental setup

In our experiments, we compared our approach (Fig. 4) to the approach that uses the description of bug reports and the random approach which choses severity randomly in proportion to the different classes. It is important to mention that in Eclipse and Gnome stack traces are embedded in the description. In this Section, we describe the dataset used in the experiment and provide some statistical analysis regarding the dataset.

4.1.1. The datasets

The datasets used in this paper consists of bug reports of the Eclipse and Gnome bug repositories. Eclipse is a mature dataset, which makes it suitable for mining. It has a 16 years history of bug reports and was the subject of mining challenge in the Mining Software Repositories 2008 conference. Gnome desktop is a suit of projects which provide a mature dataset with over 17 years of bug reports. It was the subject of challenge in the Mining Software Repositories 2009 conference. Both datasets are used extensively in different studies [2,27,29,18].

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. [11], and for Gnome, we use the regular expression showed in Figs. 2 and 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 labeled '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. [1,10,13]) we remove bug reports with normal severity. Furthermore, Enhancements are not considered as bug reports, and are removed from datasets when performing the experiments [18].

Triagers analyze bugs in the bug tracking system to assess the validity of bug report fields. Since users are not expert in the software system, triagers further adjust and reassign bug reports fields based on the bug report description provided by the user. Consistent to the previous studies [9,10,13,23,26,27,29], we use the severity of bugs from bug tracking system as label. These severities are set by users and then further adjusted by bug triagers, after the bug is reported by the user. Fig. 5 shows the process of assessing bug report severities by the triagers after the bug is submitted by the user [21].

We used the Eclipse dataset for the period of October 2001 to February 2015, having a total of 455,700 bug reports of which 297,151 had Normal severity and 66,873 were labeled with Enhancement severity. Also 1000 bug reports were removed from the dataset history by the Eclipse Bugzilla administrators for maintenance purposes. After removing bug reports with Normal and Enhancement severity, we have 90,676 bug reports with severities other than Normal or Enhancement. After applying regular expression, we had 11,925 bug reports having at least one stack trace in the description. Stack traces with less than three functions are removed since they are partial stack traces and may mislead the approach. The resulting dataset contains a total of 11,825 bug reports having 17,695 stack traces in their description.

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. Also 54,500 bug reports were later removed from 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 trace, we had 153,343 bug reports with at least one stack trace in their description.

4.1.2. Dataset setup

In our approach, we sort bug reports by their creation date and use the first 10% of the datasets to train the linear combination model of Eq. (1) and optimize the coefficients using Algorithm 2. We tested different training dataset sizes for both datasets and did not find noticeable

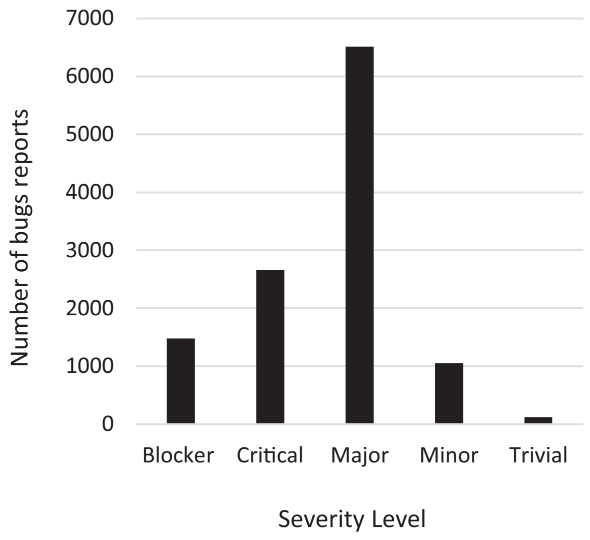

Fig. 6. Distribution of the severity labels in Eclipse dataset.

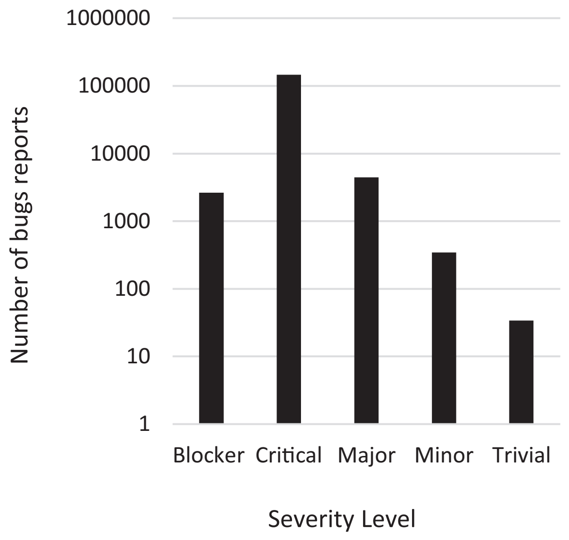

Fig. 7. Distribution of the severity labels in Gnome dataset.

differences for coefficients of Eq. (1) by increasing training dataset size beyond 10% of the bug reports. After optimizing the coefficient, we used the remaining 90% of the dataset for testing our online approach as explained in Section 3.5.1. According to our method, bug reports are sorted based on the creation date in the bug tracking system, 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 test 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 *M'th* bug report ($0 < M \leq =N$), 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 Eq. (1). The K nearest neighbor is then used to predict the severity of the *M'*th incoming bug report.

Since bug reports follow a temporal order based on their creation date, we did not use the traditional 10-fold cross-validation to validate our model. To be practical in the sense that it can be deployed in an actual bug tracking system, we create an ordered list of bug reports based on their creation date in the bug tracking system. When predicting the severity of an incoming bug report, we compare it using our linear model to bug reports that were reported before the incoming bug report only.

4.1.3. *Dataset analysis*

The distribution of severities of bug 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 Figs. 6 and 7. Note that for Gnome, the severity label distribution is shown using the logarithmic scale since the dataset is much more unbalanced 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 unbalance 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 unbalanced dataset makes it biased toward 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 unbalance dataset problem. These approaches include oversampling the minority class, under-sampling the majority class or creating cost sensitive classifier [30]. We experimented with all these approaches and observed that cost-sensitive learning [30] is the most suitable approach to overcome the unbalanced dataset problem in Eclipse and Gnome datasets.

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 don't 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 Eq. (10), the K nearest neighbor returns weight for each label, then the label with the largest weight is chosen. To make our classifier cost sensitive, in the first step, we 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 $p_1 \dots p_m$, 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., $p_1 + p_2 + \dots + p_m = 1$), the weights need to be normalized.

To calculate the probability of each label, considering that the output of our classifier is $w_1 \dots w_m$, we use Eq. (11) [18], where $W = w_1 + \dots + w_m$.

$$p_i = \frac{w_i}{W} \tag{11}$$

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 [8] in Fig. 8.

In the confusion matrix, higher values of true positives and true negatives are favorable, thus we set the misclassification cost of these two

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

Fig. 8. Confusion matrix.

values to zero. However, the cost of false positives and false negatives is selected based on the classification context.

We overcome the unbalance 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 Eq. (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 Eq. (12) [18].

$$MC_j = \frac{S}{S_j} \tag{12}$$

The cost matrix is constructed using the misclassifications cost based on Eq. (12). Then, we need to calculate the classification cost of each class label based on the cost matrix and the calculated probabilities based on Eq. (11). Assume we have M classes and the incoming bug belongs to each of these classes with probabilities of $P_1 \ldots P_m$, and assume that each class has a misclassification cost of $CO_1 \ldots CO_m$, then the cost of assigning the bug report to each of those classes is calculated by Eq. (13) [16].

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

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 Eq. (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 Eq. (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 on Eq. (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 Eq. (13), the classification cost of all other labels will be raised to $1 \times 100 = =100$. 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 Eq. (13), the classification cost of all other severity labels will be increased by $10 \times 10 = =100$, 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 take into account 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 (Fig. 9 describes this process).

4.3. Predicting severity of bugs using description

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, just as we did in our approach (Section 3.3.1). In this approach each bug report description is represented by a vector built based on the frequency of occurrence of each word (Eq. (4)) multiplied by inverse document frequency of each word (Eq. (5)). We follow the same online severity prediction approach, discussed in Section 3.5.1 for predicting severity based on weighed 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 weighed feature vectors. After calculating the distance of each bug re-

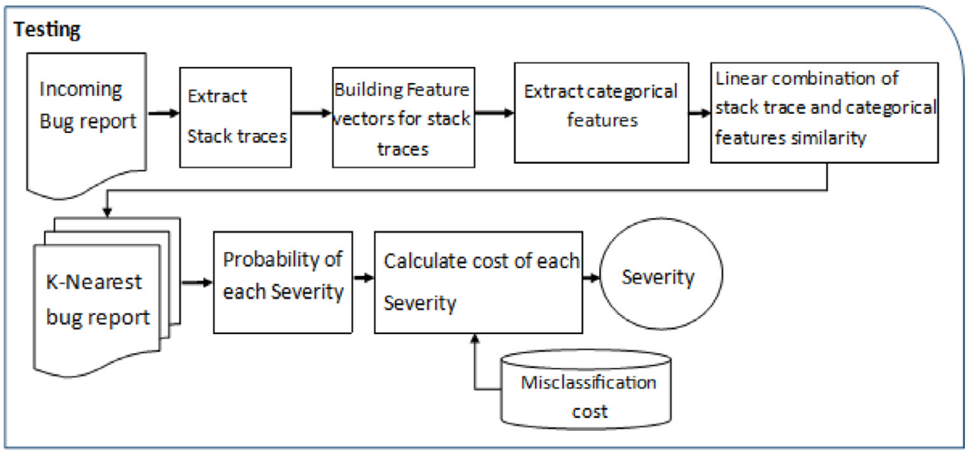
Testing
Incoming Bug report
Extract Stack traces
Building Feature vectors for stack traces
Extract categorical features
Linear combination of stack trace and categorical features similarity
K-Nearest bug report
Probability of each Severity
Calculate cost of each Severity
Severity
Misclassification cost

Fig. 9. Updated testing phase using cost sensitive k nearest neighbor.

port to all the previous bug reports, we use the K nearest neighbor bug reports to determine the bug severity label using Eq. (10). Based on the K nearest neighbor of Eq. (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 unbalance 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 approach. In this approach after calculating the probability of each severity label, we use cost sensitive k nearest neighbor of Eq. (13) which calculates the cost of choosing each severity label by considering the misclassification cost of each severity label calculated using Eq. (12) to predict severity of bug report. For example, assume our test set contains sorted bug reports $\{B_M, \ldots, B_N\}$, to predict severity of B_J $(M < J < N)$, we compared B_J description to all the bug reports in the set $\{B_M, \ldots, B_{J-1}\}$, then we predict severity of B_J using the Eq. (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 N severity classes and the number of bug reports that belong to each class is $B_{s_1} \ldots B_{s_N}$. If we randomly select severity labels for each bug, then our accuracy of predicting the severity label S_i can be calculate using Eq. (14):

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

4.5. Severity prediction approaches setup

For all the models, we followed the same approach to address the unbalance dataset distribution problem. We explained the heuristic used for choosing misclassification costs in Section 4.2. For all the models, the returned list size of the K nearest neighbor varies from 1 to 10 to assess the severity prediction accuracy with different returned list sizes. For the approach that use stack traces and categorical features, we initialized the four weights (coefficients initial value) from a normal distribution with zero mean and a standard deviation of 0.1 and used 0.001 as the learning rate. We also trained the model using the first 10% of the dataset as explained in Section 4.1.2.

4.6. 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 the precision is defined by Eq. (15):

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

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

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

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 [7]. F-measure is calculated according to Eq. (17).

$$F - measure = \frac{2\ \times Precision\ \times Recall}{Precision + Recall} \tag{17}$$

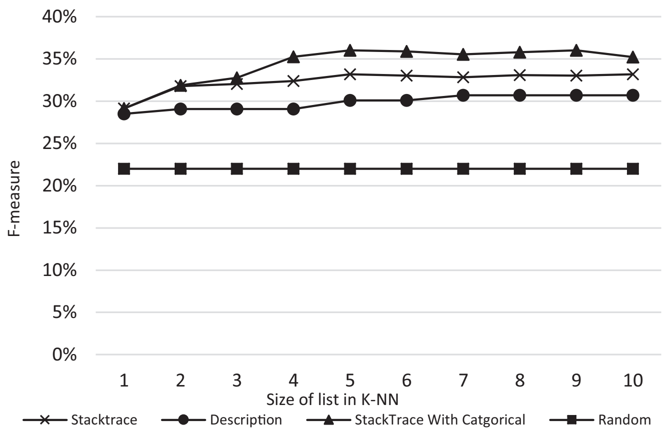

Fig. 10. F-measure of predicting Critical severity levels by varying list size in Eclipse dataset.

4.7. 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 [18], we showed that BSP_{ST} outperforms BSP_{DE}. In this study, we show how BSP_{ST+CF} performs better compared to 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.

Figs. 10–19 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 Figs. 10–19.

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 (Table A1 - A10).

RQ1) F-measure of BSP_{ST}

Figs. 10–19 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 [18], 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.

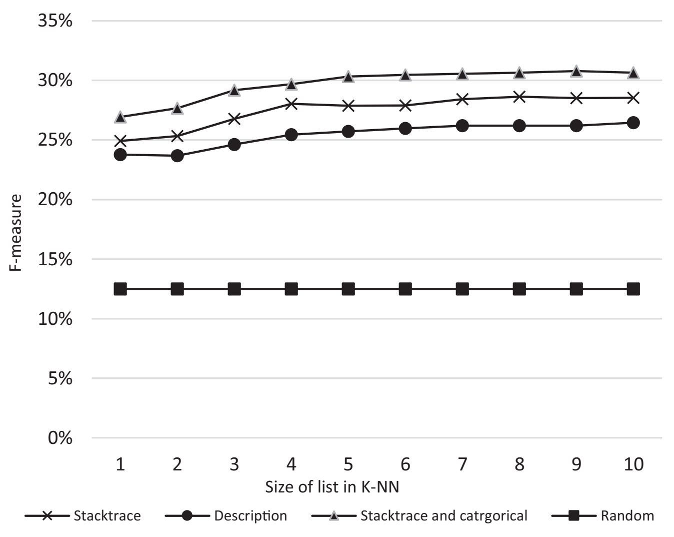

Fig. 11. F-measure of predicting Blocker severity levels by varying list size in Eclipse dataset.

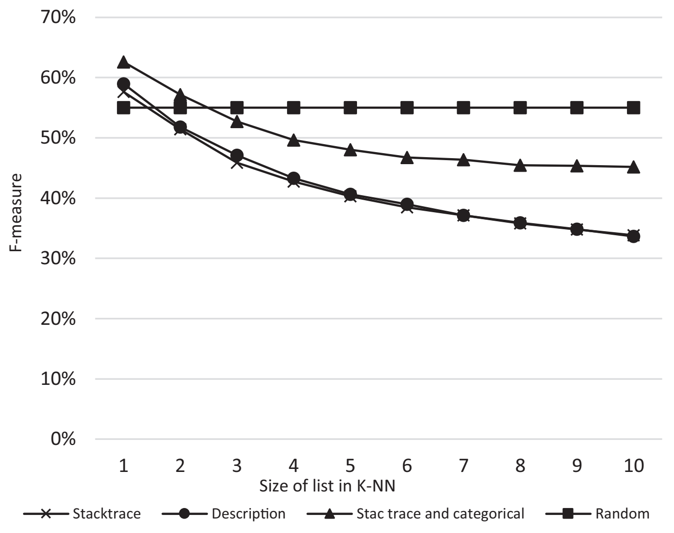

Fig. 12. F-measure of predicting Majority severity levels by varying list size in Eclipse dataset.

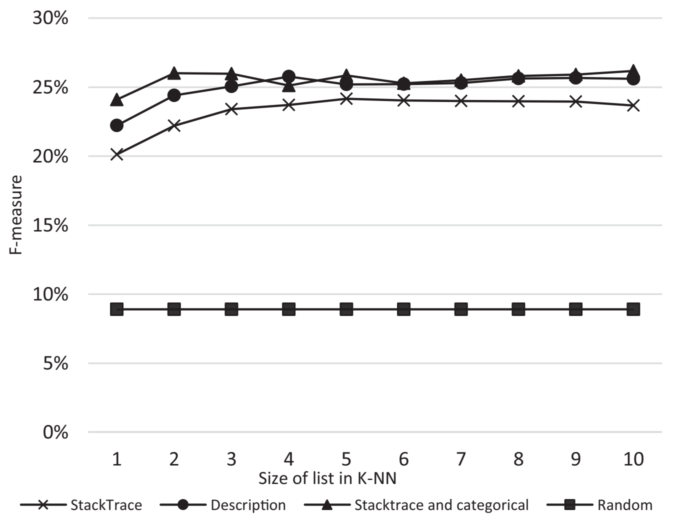

Fig. 13. F-measure of predicting Minor severity levels by varying list size in Eclipse dataset.

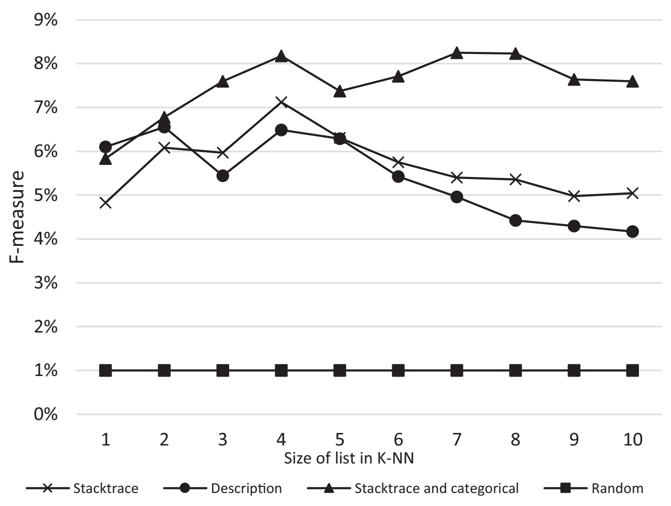

Fig. 14. F-measure of predicting Trivial severity levels by varying list size in Eclipse dataset.

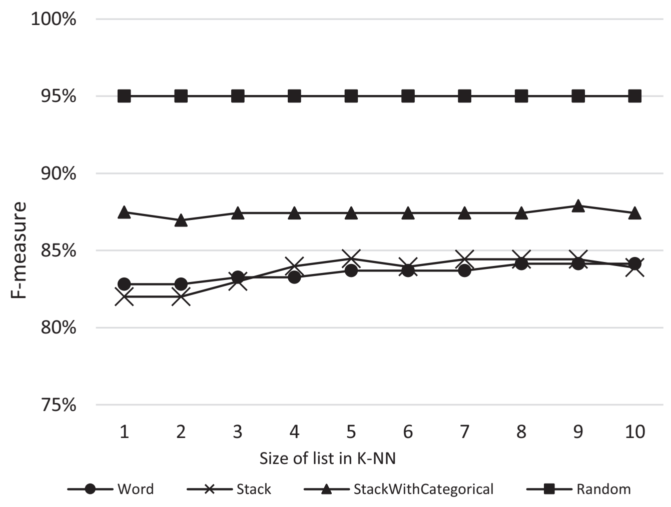

Fig. 15. F-measure of predicting Critical severity levels by varying list size in the Gnome dataset.

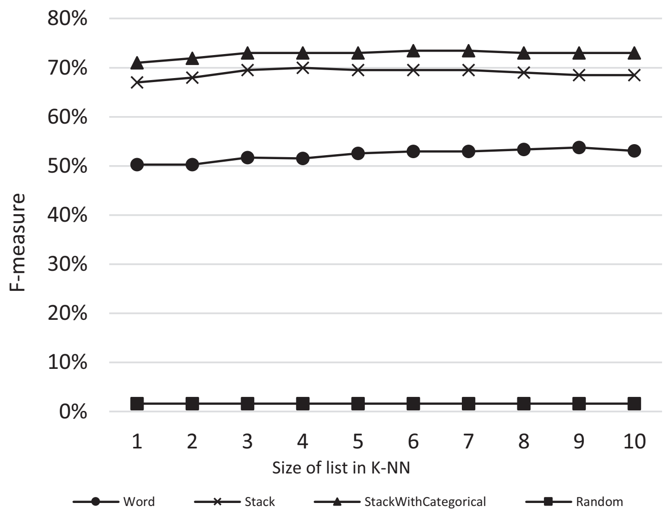

Fig. 16. F-measure of predicting Blocker severity levels by varying list size in the Gnome dataset.

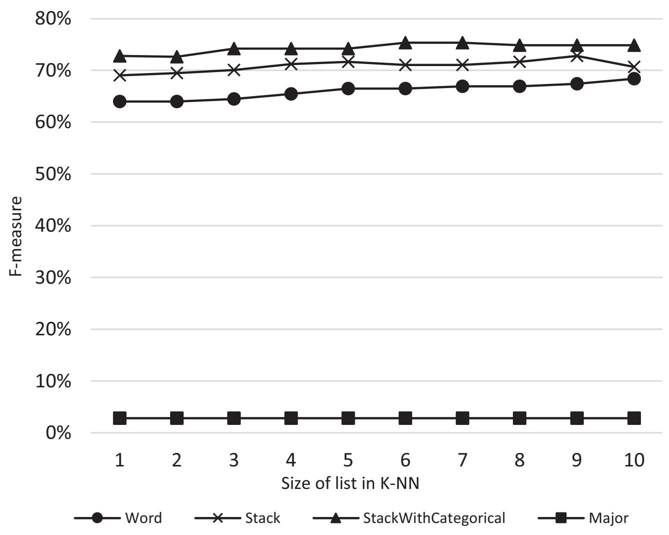

Fig. 17. F-measure of predicting Major severity levels by varying list size in the Gnome dataset.

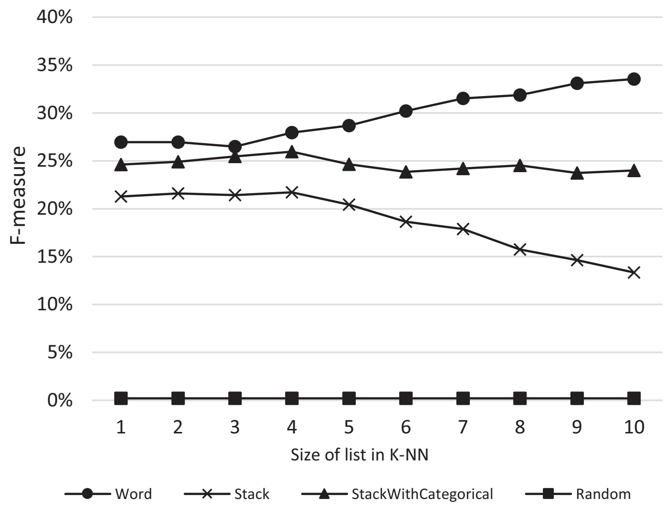

Fig. 18. F-measure of predicting Minor severity levels by varying list size in the Gnome dataset.

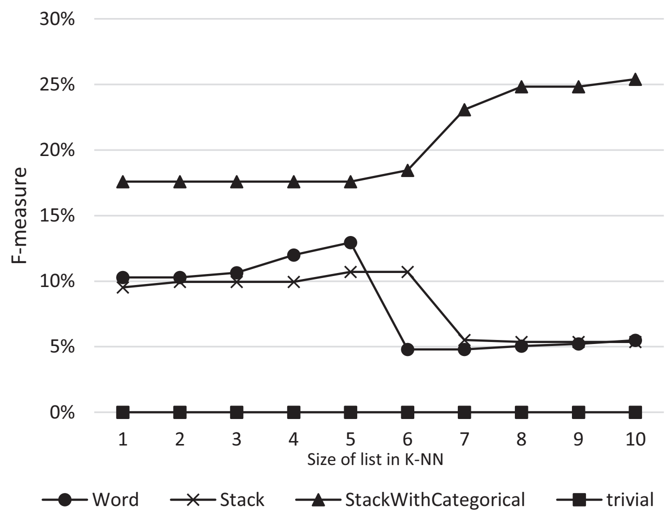

Fig. 19. F-measure of predicting Trivial severity levels by varying list size in the Gnome dataset.

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 Figs. 10–19, 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 Fig. 7, the number of bugs of Critical severity is over 100,000, which is considerably higher than any other class.

Based on Fig. 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 Fig. 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 Figs. 10–19, 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 Fig. 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 Figs. 10–14, for Eclipse dataset, we have the best performance using BSP_{ST+CF} compared to BSP_{ST} and BSP_{DE}. Also based on Figs. 15–19 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. Based on Figs. 10–19, we can conclude that using the categorical features including 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% in Eclipse to 20% in 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 Figs. 10–19 confirm that combining product, component, and operating system categorical features with stack traces increase the severity prediction accuracy.

We used two tailed Mann-Whitney test to further assess the statistical significance of difference between the results shown in Figs. 10–19. More precisely, we compare 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 detailed precision and recall provided in the Appendix A, BSP_{ST} and BSP_{ST+CF} compare to BSP_{DE} improve precision more than recall. Higher precision means that predicted severity level using BSP_{ST} or BSP_{ST+CF} compared to BSP_{DE} has higher probability of being the correct severity level. This makes the approach very suitable for developers who are reviewing the bug reports, since it helps them to properly prioritize bugs and work on critical bugs sooner.

Furthermore, we used Cliff's non-parametric effect size measure, which shows the magnitude of effect size of difference between two set of F-measures. The effect size estimates the probability that a value

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

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 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

chosen from one group is statistically higher than a value chosen from another group [6,17]. 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 [6,17]:

$$Cliff'seffectsize = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 * n_2} \tag{18}$$

In Eq. (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 [6,17]:

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

Based on Eq. (19), we can define Cliff's effect size as follows [6,17]:

$$Cliff'seffectsize = \frac{\sum_i \sum_j d_{ij}}{n_1 * n_2} \tag{20}$$

Cliff's effect size Eqn 20 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 $d \geq 0.474$ [6,17]:

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.

Tables 4 and 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 [18] as shown in the Fig. 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 Fig. 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.

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 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 the investigation, we came up with 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

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.

significantly varies from one to another. Studies [11,19] 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 [18], 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 means less similar stack traces and hence 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 is being used to predict severity too. These categorical features boost severity prediction accuracy of bug report descriptions. For instance, bug report#296383 and bug report#313534 of Eclipse bug repository are shown in Table 6 and Table 7 respectively.

In both bug reports, JPA, the faulty component of the bug, appeared in the header of the bug reports. This extra information provide a ground to make predictions based on BSP_{DE} to outperform predictions based on BSP_{ST},. However, as shown in Fig. 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 (Figs. 15–19), 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 bug's package and 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 cause the approach which uses descriptions to outperform the approach which uses stack traces and categorical features when predicting the Minor severity.

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. We also used history of bug reports in those datasets from the time of the creation of those bug repositories until 2015. 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, in Eclipse stack traces are stored in the description of the bug reports and are optional. Only 10% of Eclipse bug reports contain stack traces. Even though, Eclipse established automatic stack trace collection system since 2015, the stack traces are not publicly available yet. On the other hand, in Gnome dataset, in which same as the Eclipse bug repository stack traces are stored in the description, 45% of bug reports contain stack traces.

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 Eq. (12) to calculate the misclassification cost. This misclassification cost equation was derived based on heuristics. While the results obtained using Eq. (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. [1] 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 [3]. The authors used words in the descriptions as features. They used frequency of words for weighing the feature vector which corresponds

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 min (depends how much you use the ui)

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 [1].

Menzies et al. [13] 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. Fur-

Table 9
Gnome bug report#532680.

Bug report field	Value
Product	GIMP
Component	Plugins
Header	help-browser segfaults on 64bit systems
Description	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.

thermore, 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. [9] 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. [10] 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. [9]. 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. [26] 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. [23] 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. [25] 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. [18], they did not study the effectiveness of using the content of stack traces for predicting severity of bugs.

Yang et al. [27] 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. [2] 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. [29] 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 [20]. Sabor et al. [18] 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 [5]. Since the distribution of labels was not balance, they used a cost sensitive K nearest neighbor [16] 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.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y. Guéhéneuc, Is it a bug or an enhancement?: a text-based approach to classify change requests, in: M. Chechik, M. Vigder, D. Stewart (Eds.), Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08), ACM, New York, NY, USA, 2008 p 15, doi:10.1145/1463788.1463819.

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

[3] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, T. Zimmermann, What makes a good bug report? in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT'08/FSE-16, ACM, New York, NY, USA, 2008, pp. 308–318.

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

[5] J. Gou, L. Du, Y. Zhang, T. Xiong, A new distance-weighted k nearest neighbor classifier, J. Inf. Comput. Sci. (2012) 1429–1436.

[6] G. Macbeth, E. Razumiejczyk, R. Ledesma, Cliff's delta calculator: a non-parametric effect size program for two groups of observations, Univ. Psychol. 10 (2011) 545–555.

[7] H. Valdivia Garcia, E. Shihab, Characterizing and predicting blocking bugs in open source projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR, 2014, pp. 72–81.

[8] A. Lamkanfi, S. Demeyer, Predicting reassignments of bug reports an exploratory investigation, in: Proceedings of 17th European Conference on Software Maintenance and Reengineering, March, 2013, pp. 327–330.

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

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

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

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

[13] T. Menzies, A. Marcus, Automated severity assessment of software defect reports, in: Proceedings of the International Conference on Software Maintenance (ICSM'08), 2008, pp. 346–355.

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

[15] F. Provost, T. Fawcett, Data science for business: what you need to know about data mining and data-analytic thinking, O'Reilly Media (2013).

[16] Z. Qin, A.T. Wang, C. Zhang, S. Zhang, Cost-Sensitive classification with k nearest neighbors, in: Proceedings of 6th International Conference on Knowledge Science, Engineering and Management (KSEM'13), 2013, pp. 112–131.

[17] R. Grissom, J. Kim, Effect Sizes for Research: A Broad Practical Approach (2005).

[18] K.K. Sabor, M. Hamdaqa, A. Hamou-Lhadj, Automatic prediction of the severity of bugs using stack traces, in: B. Jones (Ed.), Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16), IBM Corp., Riverton, NJ, USA, 2016, pp. 96–105.

[19] K.K. Sabor, A. Hamou-Lhadj, A. Larsson, DURFEX: a feature extraction technique for efficient detection of duplicate bug reports, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, 2017, pp. 240–250.

[20] K.K. Sabor, M. Nayrolles, A. Trabelsi, A. Hamou-Lhadj, An approach for predicting bug report fields using a neural network learning model, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Memphis, TN, 2018, pp. 232–236.

[21] G. Sharma, S. Sharma, S. Gujral, A novel way of assessing software bug severity using dictionary of critical terms, in: Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems, 70, 2015, pp. 632–639. ISSN 1877-0509, Procedia Computer Science.

[22] C. Sun, D. Lo, S.C. Khoo, J. Jiang, Towards more accurate retrieval of duplicate bug reports, in: Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, 2011, pp. 253–262, doi:10.1109/ASE.2011.6100061.

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

[24] I.H. Witten, E. Frank, M.A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2011 ISBN 9780123748560, doi:10.1016/B978-0-12-374856-0.00021-3.

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

[26] C.-.Z. Yang, C.-.C. Hou, W.-.C. Kao, I.-.X. Chen, An empirical study on improving severity prediction of defect reports using feature selection, in: Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference – Volume 01 (APSEC '12), 2012, pp. 240–249.

[27] G. Yang, T. Zhang, B. Lee, Towards semi-automatic bug triage and severity prediction based on topic model and multi feature of bug reports, in: Proceedings of the 38th Annual Computer Software and Applications Conference (COMPSAC'14), 2014, pp. 97–106.

[28] T. Zhang, J. Chen, G. Yang, B. Lee, X. Luo, Towards more accurate severity prediction and fixer recommendation of software bugs, J. Syst. Softw. 117 (2016) 166–184 ISSN 0164-1212, doi:10.1016/j.jss.2016.02.034.

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

[30] J. Zhang, I. Mani. KNN approach to unbalanced data distributions: a case study involving information, 2003.