Empir Software Eng
DOI 10.1007/s10664-015-9409-1

On the unreliability of bug severity data

Yuan Tian[1] · Nasir Ali[2] · David Lo[1] · Ahmed E. Hassan[3]

Abstract Severity levels, e.g., critical and minor, of bugs are often used to prioritize development efforts. Prior research efforts have proposed approaches to automatically assign the severity label to a bug report. All prior efforts verify the accuracy of their approaches using human-assigned bug reports data that is stored in software repositories. However, all prior efforts assume that such human-assigned data is reliable. Hence a perfect automated approach should be able to assign the same severity label as in the repository – achieving a 100% accuracy. Looking at duplicate bug reports (i.e., reports referring to the same problem) from three open-source software systems (OpenOffice, Mozilla, and Eclipse), we find that around 51 % of the duplicate bug reports have inconsistent human-assigned severity labels even though they refer to the same software problem. While our results do indicate that duplicate bug reports have unreliable severity labels, we believe that they send warning signals about the reliability of the full bug severity data (i.e., including non-duplicate reports). Future research efforts should explore if our findings generalize to the full dataset. Moreover, they should factor in the unreliable nature of the bug severity data. Given the unreliable nature of the severity data, classical metrics to assess the accuracy of models/learners should

Communicated by: Andreas Zeller

✉ Yuan Tian
yuan.tian.2012@phdis.smu.edu.sg

Nasir Ali
nasir.ali@uwaterloo.ca

David Lo
davidlo@smu.edu.sg

Ahmed E. Hassan
ahmed@cs.queensu.ca

[1] School of Information Systems, Singapore Management University, Singapore, Singapore

[2] Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada

[3] Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Canada

Published online: 27 October 2015

not be used for assessing the accuracy of approaches for automated assigning severity label. Hence, we propose a new approach to assess the performance of such models. Our new assessment approach shows that current automated approaches perform well – 77-86 % agreement with human-assigned severity labels.

Keywords Data quality · Bug report management · Severity prediction · Performance evaluation · Noise prediction

1 Introduction

Bug tracking systems allow reporters to indicate the severity of a reported bug. In practice, as highlighted by our survey (see Section 2.3), developers use these severity levels (e.g., blocker, critical, major, normal, minor, and trivial in Bugzilla) to prioritize bugs and estimate impact of bugs. For example, bugs of critical severity have higher priority and need to be fixed immediately in comparison to bugs with minor severity. Over the past decade, researchers have proposed several automated approaches to automatically assign a severity label to a newly-reported bug report (Menzies and Marcus 2008; Lamkanfi et al. 2010; Tian et al. 2012a).

To automatically assign a severity label to a bug report, a classifier is trained using information derived from historical bug reports that are stored in software repositories. The performance of the classifier is measured by its accuracy on unseen historical bug reports that have been assigned by humans.

A key point lies in the reliability of the data used in the training and testing of such models. If the data is not reliable then the calculated accuracy of the models is unreliable as well.

In this paper, we hypothesize that assigning a severity label to a bug report is a very subjective process – i.e., for the same bug different individuals are very likely to assign it with different severity labels. Hence, we believe that bug severity data is not reliable. Such unreliable data negatively impacts the accuracy of many research approaches and leads to incorrect conclusions or findings. For instance, if the labels of the training data are not reliable, the models that are learned from such data might not be able to correctly assign labels to the testing data (Kim et al. 2011). Motivated by the potential impact of reliability of the severity labels for research study and development process, we conduct this study.

However, little is known about the reliability of the severity labels for bug reports – i.e., what is the probability that two individuals would assign the same severity label to the same software issue. To study the reliability of severity data in bug reports, we turn our attention to duplicate bug reports. While there has been a slew of studies (Runeson et al. 2007; Sun et al. 2010; Nguyen et al. 2012; Tian et al. 2012b) which attempt to identify duplicate bug reports, none of the prior studies have used duplicate reports to better understand the challenges of automatically assigning severity levels to bug reports. In particular, we examine the following research questions:

RQ1: How reliable are human-assigned severity labels for duplicate bug reports?

We investigate bug reports from three software systems, i.e., OpenOffice, Mozilla, and Eclipse, which have been flagged as duplicates of other reports. Bug reports are marked as duplicates by bug triagers, i.e., expert developers that assign bug reports to developers to

be fixed, if they describe symptoms of the same bug. We refer to a set of bug reports that are duplicate of one another as a duplicate *bucket*. Our investigation finds that up to 51 % of the duplicate buckets, have inconsistent severity labels, i.e., duplicate reports referring to the same problem have different severity labels.

We manually examine a statistical-representative sample of the duplicate buckets (1,394 bug reports across 437 bucket), in order to verify whether bug reports in the same bucket describe the same problem (i.e., whether they are truly duplicates). We find that 95 % (+/- 3 %) of the buckets contain bug reports that describe the same problem. The rest of the buckets (around 5 %) have at least one bug report that is not an exact duplicate. For example, a bucket might have two bug reports (A and B) with B containing A (i.e., bug report B describes a super set of problems described in A).

Hence, studies which automatically assign severity levels to bug reports should at minimum remove duplicate reports with inconsistent severity labels from their dataset otherwise they are using very unreliable, i.e., noisy data. We believe that our observation sends warning signals about the reliability of the full bug severity data, i.e., including non-duplicate reports. However, more studies are required to explore if our findings generalize to the full bug severity data. Moreover, our findings highlight the challenge associated with assigning a severity level for a bug report by a human and even by automated techniques – i.e., if humans cannot agree on the severity of a particular problem then we should not expect automated models to perform any better!

RQ2: How robust are classifiers against unreliable labels?

Looking at various state of the art automated classifiers, we find that as we increase the unreliability of the bug severity labels, the accuracy of classifiers suffers. Some classifiers appear to be more robust when dealing with such unreliable data. In particular, a nearest neighbor based classifier (REP+kNN) (Tian et al. 2012a) and a Support Vector Machine (SVM) classifier are more robust than decision Tree (DT) and Naive Bayes Multinomial (NBM) classifiers.

We then sought to create a new approach to better assess the accuracy of automated approaches for assigning severity labels given the unreliable nature of the data. Traditional approaches to assess model/classifier accuracy assume that the used data is of high reliability. Our new assessment approach models the problem of assigning severity labels as a multi-rater problem, then we can make use of traditional metrics to assess inter-rater agreement. We measure the inter-rater agreement between humans (on duplicate bug reports) and between humans and automated classifiers. Our new assessment approach shows that current automated approaches perform well – 77-86 % agreement with human-assigned severity labels.

The main contributions of this work are as follows:

1. **Bug severity data is noisy:** Our analysis of duplicate bug reports shows that most duplicates (95 %) correspond to the same problem. Yet, more than half of the duplicate buckets (51 %) do not have the same severity labels.
2. **Automated assignment of severity labels has high agreement with human-assigned labels:** The use of traditional approaches (e.g., precision and recall) to assess automated classifiers should be avoided given the noisy nature of the ground truth, instead inter-rater agreement measures should be used. We show that current automated approaches have a 77-86 % agreement with human-assigned severity labels.

The structure of this paper is as follows. Section 2 introduces how severity information is used in multiple dimensions. Section 3 introduces a general framework for approaches that assign severity labels for bug reports. Sections 4 and 5 provide the details on our empirical study and its results. Section 6 provides the details on our new classifier assessment approach. We summarize related work in Section 8. Section 9 concludes the paper.

2 Motivation

In this section, we motivate our empirical study on severity level of reported bugs by investigating how severity information is managed in bug tracking systems, used in prior research studies, and leveraged by developers in bug fixing process.

2.1 Severity Information in Bug Tracking Systems

Bug tracking systems are adopted by many developers to collect bug reports and manage reported bugs. Commonly used bug tracking systems, such as Bugzilla and JIRA, provide a field to capture the severity level of a reported bug. Bugzilla provides a "severity" field and it can typically take one of the following values: blocker, critical, major, normal, minor, and trivial (some projects might have less values). JIRA provides a "priority" field which serves the same purpose as Bugzilla's "severity" field. When a bug reporter submits a bug report, he/she also specifies the severity of the reported bug. In almost all cases (i.e., more than 90 %), the value of this field remains unchanged during the triaging and resolution of a bug report Xia et al. (2014). As done in prior research, we do not consider the severity label "normal" since some bug reporters just assign the label "normal" to all bug reports by default without careful consideration (Lamkanfi et al. 2010, 2011; Tian et al. 2012a).

2.2 Severity Information in Prior Research Studies

Besides studies that propose automated severity prediction tools (Menzies and Marcus 2008; Lamkanfi et al. 2010, 2011; Tian et al. 2012a), the severity level of a reported bug has also been considered in many other research studies (Zhang et al. 2012, 2013; Thung et al. 2012, Shibaba et al. 2013; Valdivia Garcia and Shibab 2014; Tian et al. 2014). Some of them investigate the relationship between bug severity level and delay in bug reporting and fixing process (Zhang et al. 2012; Thung et al. 2012). They find that severity level of a bug report is significantly related to the length of the time interval between a bug being assigned to a developer and the developer starts to fix the bug, but is not related to the length of the time interval between a bug being introduced into the code and the time it is reported. Other research studies make use of severity information to predict blocking bug reports (Valdivia Garcia and Shihab 2014), bug-fixing time (Zhang et al. 2013), the probability of a bug report being re-opened (Shihab et al. 2013), etc. Therefore severity field holds an important piece of information that can affect the effectiveness of these tools in helping developers manage bug reports.

2.3 Severity Information in Practice

To understand how severity information is used in practice, we conducted an online survey. The target participants of our survey are developers who have resolved bugs in a number of open source projects. We collected email addresses of developers that have resolved Eclipse,

OpenOffice, Mozilla, and Apache bugs. Eclipse, OpenOffice, and Mozilla use Bugzilla as the bug tracking system, while Apache uses JIRA. We sent around 500 personalized email to some of these developers to encourage them to participate in our survey. Within one week, we received 21 responses. Although 21 is not a large number, due to the difficulty in having a large number of developers to participate in a study, many past studies had also only surveyed or interviewed a similar number of participants (Espinha et al. 2014; Vyas et al. 2014).

Our survey includes three key questions.[1] That are relevant to this study:

– The first question asks a participant whether he or she considers the value of the "severity" field when resolving a bug report. We provide five possible responses:

  1. Always
  2. Most of the time
  3. Sometimes
  4. Rarely
  5. Never

– The second question asks how the value of the "severity" field in a bug report affects developers in resolving it. Developers can pick one or more of the following responses:

  1. It helps me assess the impact of bug reports
  2. It helps me prioritize which bug reports to work first
  3. It does not help me at all
  4. Other

– The third question asks how developers assess the severity of a collection of duplicate bug reports with different severity labels. We provide them 5 options for them to choose:

  1. I consider the severity levels of all of them.
  2. I only consider the severity level of the first reported bug.
  3. I only consider the severity level of one of the bug reports.
  4. I do not consider any of them.
  5. Other.

We summarize the results of our survey as follows:

– Ninety percent of the participants have considered the value of the "severity" field when resolving a bug. Only 2 participants reported that they never considered it. Seventy six percent of the participant consider it most of the time (i.e., 13), or always (i.e., 3).
– Sixty seven percent of the participants agree that the value of the "severity" field helps them to prioritize which bug reports to work first, while, twenty nine percent of them use it to assess the impact of bug reports.
– Around half of the participants (9 out of 21) said they consider the priority levels of all bug reports when a duplicate bug report group has inconsistent severity labels.

[1] JIRA bug reports do not contain severity field but only priority field (which has the same meaning as the severity field for Bugzilla bug reports). Thus when we sent emails to Apache developers who are using JIRA, we replace the term "severity" with "priority".

Some (i.e., 6 out of 21) chose the fifth option (i.e., Other). The remaining three options received much less votes (i.e., 1 to 3 votes).

3 Background

3.1 Automated Approaches for Assigning Severity Labels

A number of prior studies propose several approaches for assigning severity labels for bug reports (Menzies and Marcus 2008; Lamkanfi et al. 2010, 2011; Tian et al. 2012a). The underlying idea behind these prior approaches is that the severity label of a newly-reported bug report could be determined using properties learned from previous bug reports where the severity labels are already known (as they were assigned by a human).

Properties can be learned from terms that appear in the description and summary field of bug reports. For example, the reporter of a bug report might use words like "crash", "critical" to describe high severity reports. On the other hand, words like "typo", "enhancement" might suggest that the bug report is of low severity. Besides textual information, non-textual information, such as the component or product that are affected by the bug report could also impact the severity of a bug report.

Figure 1 gives an overview of the general framework used for automated assignment of bug severity labels. First, a classifier is trained using historical bug reports then the classifier is applied on a set of unseen bug reports. Results on this set of new bug reports are then used to assess the effectiveness (i.e., accuracy) of the classifier. The whole framework consists of four major steps: data preprocessing, classifier training, bug report severity assignment, and classifier evaluation. We describe below the details of each step.

Step 1: **Preprocessing the bug reports.** Before preprocessing the bug reports, we need to download raw bug reports from a bug tracking system (e.g., Bugzilla, Jira). These bug reports are usually stored in XML format and contain all contents of the fields of a bug report. Examples of often-used fields of a bug reports are: the long description provided by the reporter of the bug report, the short summary/title of the bug report, the component and product that is impacted by this bug. Therefore, we first extract information from the fields in the raw bug reports. For textual information inside description and summary

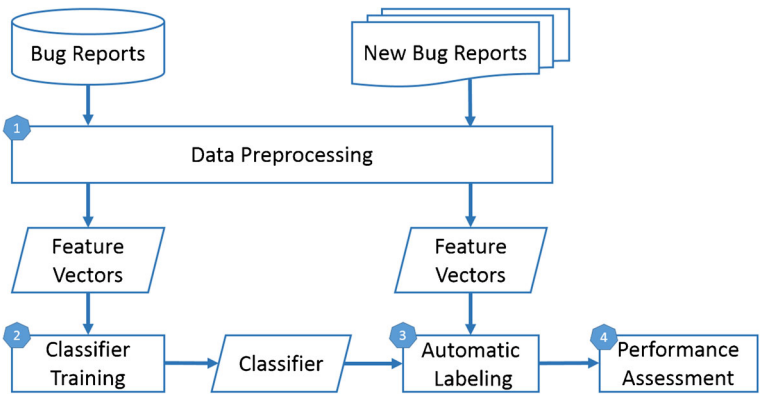

Fig. 1 Overall framework for the automated assignment of bug reports severity labels

fields, typical text preprocessing steps are applied: we tokenize sentences into tokens, we remove stop words,[2] and we stem words into their root form. These steps are helpful in removing unimportant information inside bug reports. The severity level of a bug report is also extracted. At the end of this step, we have features that are derived from both the textual and non-textual fields inside raw bug reports. These features are then used as the representation of a bug report. Each bug report is represented as a feature vector.

Step 2: Training of a classifier. Severity assignment can be regarded as a classification task. As many other classification tasks, this training phase takes feature vectors that are extracted from a set of bug reports (i.e., training data) as input. Next, a classifier learns particular properties from training bug reports. A classifier is the output of the training step.

Step 3: Automatic labeling of the severity levels. In this step, unseen bug reports are preprocessed and new unseen feature vectors are created for each bug report. These feature vectors are then inputted into the learned classifier from Step 2. For each bug report, the classifier would calculate and output a severity level which has the highest probability to be the actual severity level of the bug report.

Step 4: Assessing the performance of the classifier. This step compares the automatically assigned severity level with the actual severity level of each bug report from Step 3. Metrics such as accuracy, precision, recall, F-measure are commonly used to assess the performance of a classifier for each severity level.

3.2 Features

In the preprocessing step (Step 1), features are extracted for each bug report. In this paper, we consider both textual features and non-textual features (similar to state-of-the-art research in this area). Textual features are words that appear in description and summary of bug reports. The value of the feature is the number of times that a particular word appears in a bug report. Non-textual features are the component and product impacted by the bug. The value of a feature is a binary value (0 or 1) – stating whether that particular component or product is the one that this bug is affecting.

3.3 Classifiers

A variety of classification models from the data mining community could be used to automatically assign severity labels to bug reports. In this paper, we used different types of classifiers (e.g., tree-based versus clustering-based classifiers). We briefly discuss the various classifiers that we used in this paper.

Decision Tree (DT) A decision tree classifier is a non-parametric supervised learning method. This classifier assigns a severity label to a bug report by learning decision rules from features extracted from historical bug reports. Decision trees are very attractive to use since they are simple to understand and to interpret. They are also able to handle both numerical data like textual features and categorical data like non-textual features. However, decision tree classifier can create complex trees that do not generalize well. Hence, decision trees have a tendency to overfit, leading to bad performance on testing data. Also, decision tree classifier might create biased trees if some classes dominate others, which is the case

[2]Stop words are words (like “a” and “the”) that do not carry much specific information.

for severity data where there are much more bug reports with a normal level or a major level than other severity levels.

Naive Bayes Multinomial (NBM) NBM classifiers find their way into many applications nowadays due to their simple underlying principle yet powerful accuracy. NBM classifiers are based on a statistical principle. Specifically, each word in the preprocessed training bug report is assigned a probability that the word belongs to a certain severity level. The probability is determined by the number of occurrences of words in the bug report. For each new bug report, given words that appear in this bug report, NBM classifier could compute the sum of the probabilities for each severity level of each term occurring within the bug reports datasets, which is used to determine the severity of a bug report. Recent studies show that the Naive Bayes classifier outperforms support vector machine and simple 1-nearest neighbor classifiers when assigning severity labels for bug reports (Lamkanfi et al. 2011).

Support Vector Machine (SVM) An SVM classifier represents the features of each bug report in a high dimensionality space. In this space, an SVM classifier learns an optimal way to separate the training bug reports according to their severity labels. The output of this classifier are hyperplanes between feature vectors. The hyperplanes separate the space into multiple categories. Each category represents one severity level. Given a new bug report, SVM classifier maps the feature vector of the bug report to the subspace that it belongs to (leading to a severity level). Several studies in the data mining community show that SVM classifiers are very well suited for text categorization (Joachims 1998).

***k*-Nearest Neighbor (kNN)** A k-Nearest Neighbor classifier compares a new document to all other documents in the training dataset. The assigned label for a bug report is then based on the prominent category within the K most similar bug reports from the training dataset (the so-called neighbors). The most important point about a k-Nearest Neighbor classifier is the measurement used to compute the similarity between two bug reports. In this study, we consider our previously proposed nearest neighbor based approach in Tian et al. (2012a). In Tian et al. (2012a), the similarity between two bug reports is computed using a measurement called "REP". Given two bug reports d and q, the similarity function $\text{REP}(d, q)$ is a linear combination of four features with weights to control for the importance of each feature. These four features are based on the textual content of bug reports, as well as the component and product affected by the bug reports. In this study, we set k as 1 where the severity label of a bug report is assigned according to the severity label of the most similar report from the training set.

3.4 Evaluation Metric

Regarding the fact that bug severity predication is a general classification task, commonly used measures, such as accuracy, precision, recall and F-measure, could be taken to evaluate the performance of an automated bug severity assignment approach (Lamkanfi et al. 2010, 2011; Menzies and Marcus 2008; Tian et al. 2012a).

In this work, different from previous work, we adopt a type of inter-rater agreement based metric (i.e., Krippendorff's alpha) to evaluate the performance of an automated bug severity assignment approach. We take this measure based on two characteristics of this task:

– **Ordinal severity labels.** Precision and recall are suitable for categorical but not ordinal data. For categorical data, different classification errors can be treated equally, but it is

not the case with ordinal data. Severity labels of bug reports are ordinal data since there is an order among the different categories represented by the labels, i.e., "blocker" is more important than "critical", which is more important than "major". However, standard classification evaluation metrics cannot make use of this information. For instance, assigning a "critical" label to a "blocker" bug report is not as bad as assigning a "trivial" label to the bug report. However previously-used metrics regard these two cases as the same because they all fail to assign the actual label to the bug report. Inter-rater agreement based metrics, on the contrary, think the former case is better than the later one, as the difference between the blocker level and the critical level is less than that between the blocker level and trivial level.

– **Imbalanced data.** Precision and recall scores are potentially skewed for imbalanced data. For such data, a trivial classifier that predicts everything as the majority class will receive a high precision and recall. Usually, a bug tracking system contains many more bug reports of severity levels normal, major, or minor than bug reports of severity levels blocker, critical or trivial. Therefore, precision and recall are not suitable metrics to assess the effectiveness of a severity prediction approach. Instead, commonly used inter-rater agreement based metrics, e.g., Cohen's kappa and Krippendorff's alpha, are more appropriate since they consider *agreement by chance* that is determined by class distributions, hence, factoring in the imbalanced nature of the data.

In this paper, we choose the Krippendorff's alpha reliability (Krippendorff 2003) as the evaluation metric, rather than any other inter-rater reliability metrics. Because the Krippendorff's alpha reliability generalizes across scales of measurement, can be used with any number of observes, with or without missing data (Hayes and Krippendorff 2007). Next, we introduce the Krippendorff's alpha reliability and the methodology of calculating Krippendorff's alpha given the labels of a set of bug reports assigned by human or classifier.

Krippendorff's alpha reliability test computes an α to measure the degree of agreement among raters:

$$\alpha = 1 - \frac{D_o}{D_e} \tag{1}$$

where D_o refers to the *observed* disagreements among ratings to units of analysis and D_e stands for the *expected* disagreements when the ratings to units are randomly assigned. The value of α is less than or equal to 1 where $\alpha = 1$ indicates perfect reliability and $\alpha = 0$ means that raters assign labels randomly. The α value could also be negative when disagreements are systematic and exceed what can be expected by chance. The mathematical formulas to compute D_o and D_e are given below:

$$D_o = \frac{1}{n} \sum_{c} \sum_{k>c} CNT_{(c,k)} \times DIST_{(c,k)} \tag{2}$$

$$D_e = \frac{1}{n(n-1)} \sum_{c} \sum_{k>c} n_c \times n_k \times DIST_{(c,k)} \tag{3}$$

In the above equations, c and k are possible values of a rating. $CNT_{(c,k)}$ is a value in a *count matrix* that corresponds to the number of possible pairings of raters, e.g., r_1 and r_2, where r_1 gives a rating c and r_2 gives a rating k respectively to units of analysis. n_c and n_k corresponds to $\sum_k CNT_{(c,k)}$ and $\sum_c CNT_{(c,k)}$ respectively. $DIST_{(c,k)}$ is a value in a *distance matrix* that corresponds to the square difference between the two rating values c and k.

When we use the Krippendorff's alpha to measure the performance of a bug severity assignment approach, the unit of analysis is one bug report. For each bug report (i.e., unit),

Unit (Bug Report)	1	2	3	4	5	6	7	8	9	10
Actual Severity	1	2	3	3	4	3	3	4	3	5
Predicted Severity	2	3	3	3	3	3	3	5	3	2
#Raters	2	2	2	2	2	2	2	2	2	2

Fig. 2 An example reliability data matrix

we have two raters, one is the developer who assigns the severity level of this bug report, the other is a classifier (e.g., a decision tree classifier, a naive bayes multinomial classifier). A larger alpha represents a higher agreement between the classifier and the developer, which also means a better performance of a classifier.

The **input** data for analysis is the actual severity levels of bug reports assigned by developers and the severity labels assigned by the classifiers. The **output** is the α reliability. Before calculating the alpha value, we first map the severity labels to numerical values, e.g., 1, 2, 3, 4, and 5. We then compute the alpha value based on the four steps described below.

Step 1: Generate a reliability data matrix. Figure 2 shows an example of a reliability data matrix. In this example, we have 10 bug reports and five rating values (i.e., 1, 2, 3, 4, and 5). For each bug report u, we compute the number of ratings given by the different raters and denote them as m_u. Note that in this sample, m_u is a constant which equals to 2.

Step 2: Compute count matrix based on reliability data matrix. From the reliability data matrix, we compute a symmetric count matrix $CNT_{l\times l}$, where l is the number of rating values. The matrix is shown in Fig. 3a. A value in the matrix is denoted as $CNT_{(c,k)}$ and it is defined as:

$$CNT_{(c,k)} = \sum_{u} \frac{\#Pairs(c,k,u)}{m_u - 1} \tag{4}$$

In the above equation, c and k are rating values, u is a unit of analysis (in our case: a bug report), and m_u is the number of ratings for unit u which equals to 2. $\#Pairs(c,k,u)$ is the number of rater pairs, e.g., r_1 and r_2, where r_1 gives a rating of value c and r_2 gives a rating of value k to unit u respectively. The count matrix corresponding to the reliability data matrix given in Fig. 2 is shown in Fig. 3. In the matrix, the value of $CNT_{(3,3)}$ is 10, since 2 possible pairings of raters (i.e., (actual, predicted) and (predicted, actual)) give ratings 3 and 3 to 5 bug reports (i.e., bug reports 3, 4, 6, 7, and 9).

Step 3: Compute distance matrix. Bug report's severity level is an ordinal value, which means that the distance between rating 1 and 5 is larger than that between rating 1 and

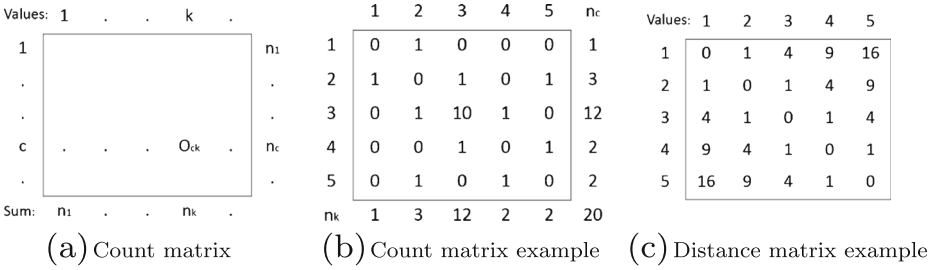

Fig. 3 Count matrix and distance matrices

2. The distance matrix $DIST_{l\times l}$, where l is the number of rating values, is a symmetric matrix which contains the square differences between each possible pairings of ratings. Each element of the matrix, denoted as $DIST_{(c,k)}$ is defined as:

$$DIST_{(c,k)} = (c-k)^2 \tag{5}$$

We show the distance matrix for ratings 1 to 5 in Fig. 3c.

Step 4: **Step 4: Compute α-reliability.** Following (1), (2), and (3), we compute the α-reliability. Using the count matrix and distance matrix shown in Fig. 3b and c, we get $\alpha = 0.27$.

4 Empirical Study Design

In our effort to better understand the reliability of bug severity data, we perform an empirical study on three datasets, i.e., OpenOffice, Mozilla, and Eclipse. In this study, we analyze the reliability of human-assigned severity labels for duplicate bug reports and the robustness of classifiers in the presence of unreliable severity labels.

4.1 Goal and Variables

The *goal* of our empirical study is two fold; first, we quantify the unreliable nature of human-assigned severity labels, second, we analyze the robustness of four state of the art classifiers. We choose the following four classifiers to assign severity labels to bug reports: 1) information based nearest neighbor classifier (denoted as REP+kNN in this paper) (Tian et al. 2012a), 2) DT classifier, 3) NBM classifier, and 4) SVM classifier. We choose these classifiers because they are commonly used to solve mining tasks in software engineering and have been used in prior studies to automatically assign severity labels to bug reports (Lim and Goel 2006; Huang et al. 2006; Tian et al. 2012a). The *perspective* is that of practitioners and researchers, interested in understanding the reliability of the human-assigned severity labels and the impact of unreliable severity labels on automated classifiers. The *objects* of our case study are three open-source software systems, i.e., OpenOffice, Mozilla, and Eclipse.

4.2 Studied Datasets

We use the same datasets as (Tian et al. 2012a). The open-source nature of the studied systems ensures that other researchers can replicate our experiments. The large size of the studied systems ensures that the systems are representative of the scale of systems that most developers would face in practice. Table 1 shows the statistics of three studied datasets. Note

Table 1 Statistics of datasets

Dataset	Period		Bug Reports		
	From	To	#All	#Duplicates	#Non-normal duplicates
OpenOffice	2008-01-02	2010-12-21	23,424	2,779	422
Mozilla	2010-01-01	2010-12-31	72,428	6,615	1,471
Eclipse	2001-10-10	2007-12-14	178,609	21,997	4,965

that the last column in Table 1 represents the duplicate bug reports that have a non-normal severity level (i.e., blocker, critical, major, minor, trivial).

5 Empirical Study Results

This section presents the results of our two research questions. For each research question, we present its motivation, used approach, evaluation metric(s), and a discussion regarding our findings.

RQ1: How reliable are human-assigned severity labels for duplicate bug reports?

Motivation Automatically assigning the severity level of a new bug report is an active research area (Lamkanfi et al. 2010, 2011; Menzies and Marcus 2008; Tian et al. 2012a). To assign a severity label to a bug report, a classifier is trained using information derived from historical bug reports, which are stored in software repositories. Human-assigned severity labels of bug reports are used to train a classifier to assign a severity label for a new bug report. The performance of a classifier highly depends on the reliability of the training data. The high dependence of classifiers on historical data raises a question about the reliability of the historical data. If the data used for training and/or testing is not reliable then the accuracy of the trained classifiers is unreliable as well. In this RQ, we attempt to quantify the reliability of historical severity data of bug reports.

Approach To quantify the reliability of human-assigned severity labels for bug reports, we consider the severity labels of duplicate bug reports. We would expect that the severity labels for duplicate reports to be the same since they refer to the same software problem.

We look at duplicate bug reports that have been manually marked as duplicated by domain experts. For each set of duplicate bug reports, we create a bucket (i.e., a group of bug reports that are reporting the same bug). We consider a bucket to be clean if all the bug reports inside the bucket have the same severity level. Otherwise, the bucket is considered a *noisy bucket* since it contains bug reports with inconsistent labels.

We analyze duplicate bug reports extracted from three large open-source software systems (OpenOffice, Mozilla, and Eclipse). As done in prior research, we do not consider the severity label normal since some bug reporters just assign the label "normal" to all bug reports by default without careful consideration (Lamkanfi et al. 2010, 2011; Tian et al. 2012a). Thus, we treat these reports as unlabeled data and do not consider them in our study.

Evaluation Metrics Intuitively, duplicate bug reports inside one bucket should be assigned the same severity level. Therefore, to investigate whether a bucket is clean or not, we consider severity labels of all bug reports inside a bucket. If all bug reports in the bucket have the same severity level, this bucket is clean and each bug report inside it is considered as a clean bug report. On the other hand, if some of the bug reports inside the bucket have different severity labels, the bug reports inside this bucket are considered as inconsistent bug reports, i.e., unreliable. Noisy buckets, i.e., where two or more humans do not agree on a single bug severity label, lead to unreliable bug severity data. After dividing all duplicate bug reports into two categories: clean and noisy, we compute some statistics for the unreliable data.

Table 2 Duplicate bug reports in OpenOffice, Mozilla, and Eclipse

Dataset	#Clean duplicates	#Inconsistent duplicates	All
OpenOffice	300 (71.09 %)	122 (28.9%)	422
Mozilla	932 (63.36 %)	539 (36.6 %)	1,471
Eclipse	2,441 (49.20 %)	2,524 (50.8 %)	4,965

Results Up to 51 % of the human-assigned bugs severity labels are unreliable. Table 2 shows the number of clean duplicate bug reports and the inconsistent duplicate bug reports in each of the studied datasets. Table 2 shows that 28.9 %, 36.6 %, and 50.8 % of the duplicate bug reports have inconsistent severity labels in the OpenOffice, Mozilla, and Eclipse datasets, respectively. While our results indicate that duplicate bug reports have unreliable severity labels, we believe that our results send warning signals about the reliability of the full bug severity data (i.e., the data that includes non-duplicate reports as well). Future research efforts should explore if our findings generalize to the full dataset. We also separately investigated the noise in each bug severity level. There are six different severity levels, i.e., blocker, critical, major, normal, minor, and trivial, in Bugzilla.

Most of the unreliable data has "major" severity label. Figure 4 shows the distributions of clean and inconsistent bug reports for each severity label for OpenOffice, Mozilla, and Eclipse. We removed the normal severity bug reports from our study. Consequently, the normal level severity bugs are 0 in our dataset. In the case of clean bug reports, OpenOffice and Eclipse have more clean bug reports in the "major" severity label than Mozilla. Only in Mozilla, clean bug reports of "critical" severity types are more than inconsistent buckets. In OpenOffice, we do not have "blocker" severity type bug reports.

Our manual analysis of duplicate buckets shows that 95 % of the buckets contain bugs that correspond to the same problem – providing support to our intuition that duplicate bugs should have the same severity labels. A developer might bug report A as a duplicate of bug report B if B contains A. In this case, two duplicate bug reports may have two different severity levels.

To verify such duplicate bug report cases, we manually examined a statistical sample to verify whether or not duplicate bug reports with inconsistent severity labels describe the same problem. We use 95 % confidence level with 5 % confidence interval to select a statistical sample for manual verification. The statistical sample contains 1,394 bug reports (in 437 buckets) that need to be manually verified. More precisely, there are 38, 127, and 272 noisy buckets containing 122, 539, and 733 bug reports of OpenOffice, Mozilla, and Eclipse respectively.

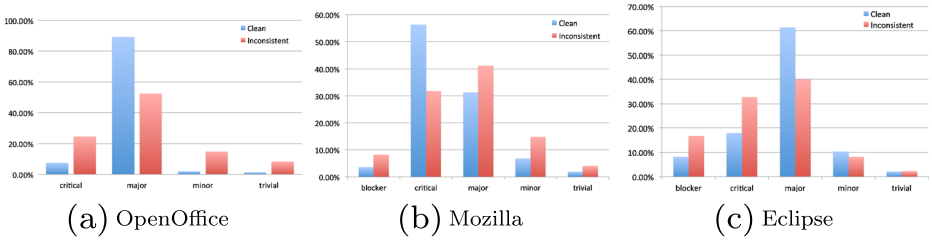

Fig. 4 Distribution of clean and inconsistent bug reports for each severity label

The first two authors manually verified all the buckets in our statistical sample. In the case of a disagreement, both authors had a meeting to resolve the disagreement. We observe five general duplicate patterns through our manual verification:

- *Inconsistent:* 95 % of the noisy buckets indeed have the same severity bugs with different severity labels (i.e., 92 % of the noisy buckets in OpenOffice, 92 % of the noisy buckets in Mozilla, and 97 % of the noisy buckets in Eclipse). Sometimes, the same reporter posted the same bug report with two different severity labels. For example, `Bug#142103` and `Bug#170098` in the Eclipse dataset.
- *Subset:* 7 out of the 437 sampled noisy buckets have bug reports that are subset(s) of other bug reports in the same bucket. For instance, bug report *B* reports a subset of problem(s) that are described in bug report *A*. The bug reports in the bucket have a reason to be labeled with different severity labels. However, we noticed that in some cases, superset bug reports had lower priority than the duplicate bug reports with subset of problems. For example, `Bug#202550` and `Bug#203013` in the Eclipse dataset.
- *Dependent:* 2 out of the 437 sampled noisy buckets have bug reports that are different but dependent on each other. For example, in the OpenOffice dataset, `Bug#112523` was introduced by the fix of `Bug#111065`.
- *Hybrid:* 5 out of the 437 sampled noisy buckets contain more than two types of relationship that are introduced above. For instance, in a bucket from the Eclipse dataset, `Bug#53473` and `Bug#55269` are inconsistent with each other. However, in the same bucket, `Bug#55272` reports a subset of the problems in `Bug#53473`.
- *Others:* 9 out of the 437 sampled noisy buckets are hard to judge whether the bug reports inside the buckets are reporting the same problem or not. Some of them are even not confirmed as duplicates by developers when they are labeled as duplicates, due to reasons such as the difficulty of reproducing the bug.

To investigate which features (i.e., textual features and non-textual features described in Section 2.1) are important for identifying noisy bug reports (i.e., unreliable data), we compute the information gain for each feature. Table 3 shows the top 10 features for Openoffice, Mozilla, and Eclipse dataset.

We notice that top features vary among different software systems. We also observe that the word "crash" is common for both Openoffice and Mozilla datasets. If we consider the

Table 3 Top features for noise prediction in OpenOffice, Mozilla, and Eclipse

Feature rank	Openoffice	Mozilla	Eclipse
1	sort	cpp	main
2	crash	Product Core	signal
3	open new	crash	lang
4	anyth	crash stat	vm 1
5	reproduc	stat	launcher main
6	find	signatur expand	java
7	new writer	com report	java hotspot
8	sort column	modul signatur	nativ
9	data	frame model	java lang
10	insert	expand	hotspot

types of the features, most of them are uni-gram word features, some of them are bi-gram word features, one of them is categorical feature ("Product Core" for Mozilla).

Increase in the unreliable data rate impacts the performance of classifiers. Generally, REP+kNN and SVM classifiers achieve better accuracy and thus are more robust than DT and NBM at various levels of unreliable data rate.

RQ2: How robust are classifiers against unreliable labels?

Motivation The results of RQ1 challenge the validity of existing automated approaches for determining the severity level of bug reports. If the data is not reliable then the reported accuracy of any classifier is unreliable as well. However, very little is known about which classifier is more robust than others when dealing with unreliable data. Thus, we pose RQ2 to analyze the robustness of classifiers against unreliable data. The results of RQ2 will help future researchers/practitioners to select robust classifiers against unreliable data for automated labeling of severity labels for bug reports.

Approach We now briefly explain the steps that we followed to create our testing and training data.

Creating Training and Testing Data To investigate the effect of data reliability on different classifiers, we first create a set of training data, which contain different percentages of unreliable data. In addition, we create testing data, which only contains clean bug reports. We then measure the effectiveness of a classifier that is learned from unreliable training data on the clean testing data. Figure 5 illustrates the process of creating testing data and training data with different unreliable data rates. As shown in Fig. 5, we create the training data and testing data in two steps, which are presented as follows.

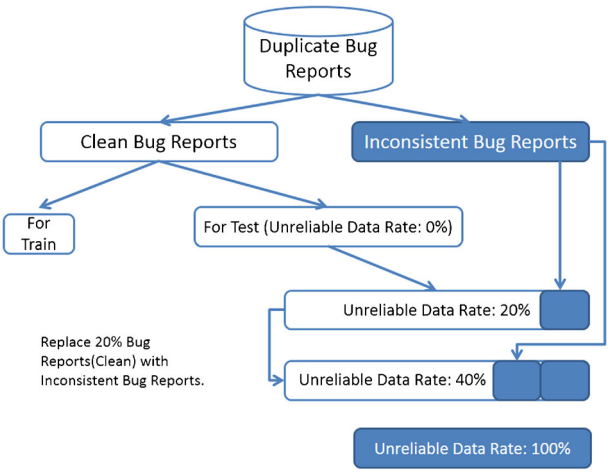

Fig. 5 Creating the training datasets and testing dataset

Step 1: **Create testing dataset.** Testing dataset contains only clean bug reports. Therefore, in the first step, after we collect both clean and inconsistent bug reports, we randomly select a 20 % of the clean data and regard them as the testing set, where the rest of the clean duplicates are used to generate a series of training data.

Step 2: **Create a set of training datasets.** To investigate the impact of unreliable data, we collect a set of training datasets where each of them contains a different percentage of inconsistent bug reports. We call the percentage of inconsistent bug reports in the dataset as *unreliable data rate*. A higher unreliable data rate means that there are more inconsistent bug reports in the dataset. Next, the training datasets are generated by randomly selecting n % of the clean training data and replacing them with unreliable data. This kind of replacement strategy to generate training data with different unreliable data rates is also used in previous work (Kim et al. 2011). By default, we set n as 20 in this study, which means that we will generate unreliable datasets by replacing 20 % of the bug reports (all clean) with unreliable bug reports. Therefore, we have six training datasets with unreliable data rates as 0 %, 20 %, 40 %, 60 %, 80 % and 100 %, respectively.

We repeat this dataset generation process (Step 2) 10 times and measure the performance of four classifiers, i.e., DT, NBM, SVM, and REP+kNN, on all testing data. Note that, in each round, the testing set remains the same for the six training sets with different unreliable data rate.

Implementation of the Four Used Classifiers We select four classifiers for our experiment. Three classifiers, i.e., DT, NBM, and SVM, are used commonly in prior research. REP+kNN has shown promising results over existing classifiers (Tian et al. 2012a). For the DT classifier, we use the J48 classifier in Weka. For NBM, we also make use of the classifier in Weka. For SVM, we use the liblinear classifier in Weka. For REP+kNN, we use the same settings as in (Tian et al. 2012a).

Evaluation Metrics We use the Krippendorff's alpha described in Section 3 to measure performance of the four classifiers on the three studied datasets. We repeat the process of generating training and testing data 10 times. We then calculate the Krippendorff's alpha across all the testing data.

Results We found that as the rate of unreliable data in training data increases, the performance of the classifiers decreases. This demonstrates that unreliable data in the training data has a negative impact on the classifier, and thus causes a worse performance on testing data.

1) *Performance on OpenOffice:* Figure 6 shows the Krippendoff's alphas of the four classifiers. Among the four classifiers, REP+kNN performs best when the rate of unreliable data is low. However, as the rate of unreliable data increases, REP+kNN performs worse than SVM. One reason might be due to the high dependence of REP+kNN on the labels of the most similar bug reports (as it uses the single most similar report). SVM is the most robust to unreliable data.
2) *Performance on Mozilla:* The Krippendoff's alphas of all classifiers decreases as the rate of unreliable data in training data increases. The results of four classifiers on Mozilla dataset are shown in Fig. 7. When unreliable data rate is smaller than 80 %, REP+kNN performs best, followed by SVM, DT and NBM. When the unreliable data rate is higher or equal than 80 %, SVM obtains the highest accuracy, followed by REP+kNN.

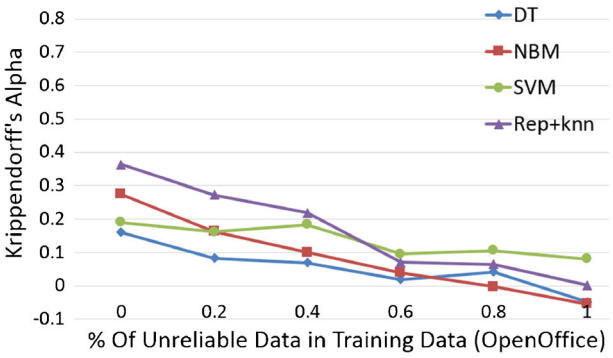

Fig. 6 Performance of the four Classifiers on OpenOffice. DT is short for Decision Tree, NBM is short for Naive Bayes Multimomial, SVM is short for Support Vector Machine, Rep+knn is short for our nearest neighbor based classifier

3) *Performance on Eclipse:* All classifiers are more stable against unreliable data compared to the OpenOffice and Mozilla datasets. The results of four classifiers on Eclipse are shown in Fig. 8. When the unreliable data rate is smaller than to 60 %, REP+kNN performs best. When unreliable data rate increases above 60 %, SVM has the highest alpha value, followed by REP+kNN and DT.
4) *Performance patterns across the three datasets:* All four classifiers are sensitive to unreliable data. However, some classifiers, at different rate of unreliable data, appear to be more robust when dealing with such unreliable data. We observe that REP+kNN performs best when the unreliable data rate is not too high (i.e., under 40 %). However its performance is impacted more severely negative than other classifiers, as we increase the unreliable data rate. SVM performs better than REP+kNN when the unreliable data rate is high.

Krippendorff advises that researchers should rely on variables with reliabilities above $\alpha = 0.80$ and consider variables with $0.667 < \alpha < 0.800$ (Krippendorff 2003). Variables with reliability less than 0.667 should be considered as not reliable. From Figs. 6–8,

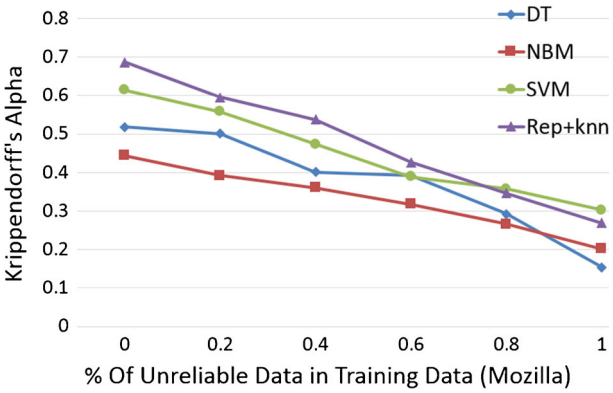

Fig. 7 Performance of the four classifiers on Mozilla. DT is short for Decision Tree, NBM is short for Naive Bayes Multimomial, SVM is short for Support Vector Machine, Rep+knn is short for our nearest neighbor based classifier

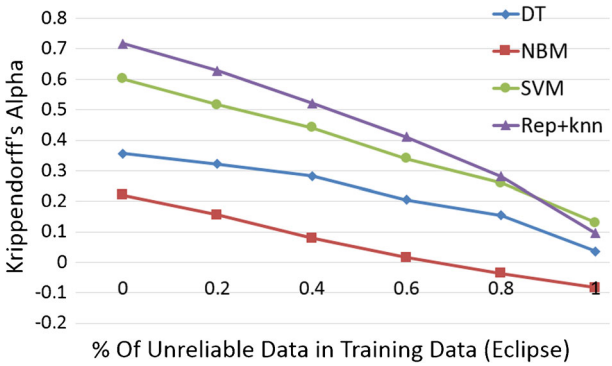

Fig. 8 Performance of the four classifiers on Eclipse. DT is short for Decision Tree, NBM is short for Naive Bayes Multimomial, SVM is short for Support Vector Machine, Rep+knn is short for our nearest neighbor based classifier

we find that a noise injection of 20% already reduces alpha below 0.667 for all cases. This shows that unreliable training data substantially adversely impacts performance of automated severity prediction approaches.

Up-to 51% of the human-assigned bug severity labels are unreliable. If humans cannot agree on the severity of a particular problem then we should not expect automatically trained classifier to perform that well either!

6 Assessing the Accuracy of a Classifier Using Unreliable Data

Our prior analysis highlights that the data used for experiments of automatically assigning severity levels to bug reports is very unreliable. Hence, the use of traditional absolute measurement metrics to assess the performance of classifiers working on such data is not advisable. Instead, the research community needs to consider other assessment approaches that factor in the unreliable nature of the data. One possible proposal, which we present in this section, is the use of *ratio* of inter-rater agreement metrics.

The idea of our newly introduced metric is, now let's forget about what is the real ground truth, because no one can get it in practice (human label is subjective), rather we consider how can a machine *act like a human which might not have the same opinion for all issues but often have similar opinions on many issues*. It allows one to live with inconsistencies (since it is likely to never go away), and assesses the performance of a severity prediction algorithm in making reasonable predictions that are as close as possible to a *reasonable range of human estimates*.

Following this idea, our new metric measures the ratio of inter-rater agreement between humans and between humans and an automated classifier – we measure the agreement between humans who assign severity labels to the same bug reports with the agreement between the human-assigned labels and the classifier-inferred labels. In particular, we use Krippendorff's alpha reliability test, which is described in Section 3, to analyze the percentage of times that humans agree on a single label for the severity level of a bug report. We

Unit (Bucket)	1	2	3	4	5	6	7	8	9	10
	1	2	3	4	5	1	3	2	2	3
	1	3	3	3	5	2	3		3	5
	1	3		2	4	2	4	2	3	5
#Raters	3	3	2	3	3	3	3	2	3	3

Fig. 9 An example of reliability data matrix

then compare the computed Krippendorff value to the value computed based on the testing data and the labels produced by an automated classifier. Our intuition is that if humans cannot agree then we cannot expect that an automated classifier can achieve 100 % agreement, instead we should seek automated classifiers that have a similar level of agreement as humans have with one another. We present below the how to calculate Krippendorff's alpha given a duplicate bug reports dataset. We then discuss the agreement ratio values for the studied projects using the best performing classifier (REP+kNN) (see Section 5).

In this case, different from using Krippendorff's alpha to measure performance of a classifier, the unit of analysis is one bug bucket (i.e., bug reports that are reporting the same bug) that contains at least two duplicate bug reports. For each bucket, the severity levels of bug reports inside the bucket become the ratings for this bucket. Severity levels are mapped to numerical values. **Input** data for reliability computation is the severity levels of bug reports in the selected buckets and the **output** is the α reliability. Note that we have multiple raters in the process, the number of raters in each bucket is different. Similar with previous section, we show a sample case in Fig. 9. Following the detailed steps described in Section 3, we could compute the Krippendorff's alpha for this sample is 0.72.

Using α-Agreement Ratio to assess automated classifiers: Results in Table 4 show the value of the reliability metric, i.e., α, for each dataset. On average, there are 2.43, 2.49, and 2.42 of raters per duplicate bug report for Eclipse, Mozilla, and OpenOffice respectively. For human judges, we consider all duplicates bug reports as a whole. For the automated classifier, REP+kNN, we separate bug reports into 10 folds and use 9 of them for training and we use the 10th fold to measure the α. In this way, we could compare the results of REP+kNN and human labels on the whole dataset.

In most of the cases, humans do not agree with each other on the severity levels of bug reports. Results in Table 4 show that most of the dataset are unreliable with respect to the severity level of bug report, except for the Mozilla dataset. The low reliability values suggest that users use different criteria when they decide the severity level of a bug report. The disagreement might be due to the users' diverse levels of experience: some users are able define the severity level of a bug report while others cannot.

We also compute the inter-rater reliability statistic, α, between the severity levels assigned by human and the one assigned by REP+kNN to investigate how our method acts compared to human judgments.

Table 4 Result of the inter-rater reliability test on duplicate bug reports

Dataset	Human	REP+kNN	Agreement ratio
`OpenOffice`	0.538	0.415	0.77
`Mozilla`	0.675	0.556	0.82
`Eclipse`	0.595	0.510	0.86

Results in Table 4 show that there is a disagreement between the labels given by human and the ones given by our automated approach. Among three datasets, Mozilla has the highest inter-rater reliability, then followed by Eclipse dataset. The reliability of severity labels on OpenOffice is the lowest. Although these three reliability values are not high enough, they are closer to the values of α on datasets judged by human. The ranks of reliability values between our approach and human judge on the three datasets are consistent (i.e., $\alpha_{Mozilla} > \alpha_{Eclipse} > \alpha_{OpenOffice}$).

The agreement ratio of α between our approach and human judgment gets 0.77, 0.82, and 0.86 on the OpenOffice, Mozilla, and Eclipse datasets respectively. This result shows that we can achieve 77-86 % of the reliability of human judgment and the performance is relatively stable among different datasets.

The Krippendorff αs for REP+kNN in Table 4 are better than some αs for REP+kNN in Figs. 6–8 (Section 5). This difference is due to the difference in the experimental setting. In this section, we use all duplicate bug reports (without selection), while in Section 5, we select duplicate bug reports to vary the percentage of unreliable data from 0 to 100 %.

7 Discussion

Threats to Validity One threat to the validity of our study relates to the generalizability of the findings. We have analyzed duplicate bug reports from three software systems and survey 21 developers. These bug reports and developers may not be representative of all duplicate bug reports and developers. Still, we have analyzed more than 1,000 duplicate bug reports. Moreover, although 21 developers is not a lot of developers, due to the difficulty in having a large number of developers to participate in a study, many past studies had also only surveyed or interviewed a similar number of participants (Espinha et al. 2014; Vyas et al. 2014). Another threat relates to experimenter bias. Our study involves a manual investigation of more than a 1,000 duplicate bug reports to check if bug reports in the same bucket describe the same problems (see Section 5, RQ1). Like other similar studies which involve manual analysis, there can be errors or subjectivity in the process.

Inconsistencies and Usage of Severity Levels Developers care about severity levels (see Section 2.3) albeit inconsistencies among severity levels assigned by reporters (see Section 5, RQ1). Assigning severity levels is partly subjective, just like when people assign ratings to hotels, movies, etc. Still, these ratings are useful, and companies use all of such ratings (which are typically different from one another) to get a larger picture of how various users perceive a certain item. Similarly, by considering all severity levels, developers can better assess the severity of a bug considering the perspectives of many people who use the system and are affected by a bug in one degree or another. Our findings suggest that developers deal well with the unreliability of severity levels and seem to only care about the ballpark figure of severity levels estimated from several reporters' inputs.

8 Related Work

8.1 Automated Classification of Bug Reports Severity Labels

Severity of a bug indicates the importance of a bug and a developer can allocate the resource accordingly. Several approaches have been proposed to automatically assign a severity level to a bug report (Lamkanfi et al. 2010, 2011; Menzies and Marcus 2008; Tian et al.2012a).

Menzies and Marcus propose a rule learning technique to automatically assign severity levels to bug reports (Menzies and Marcus 2008). They extract word tokens from bug reports, and then perform stop word removal and stemming. Important tokens are then identified using the concept of term frequency-inverse document frequency (tf-idf),[3] and information gain. These tokens are then used as features for a classification approach named the Ripper rule learner (Cohen 1995). Their approach is able to assign bug report labels on a NASA dataset.

Lamkanfi et al. use text mining algorithms to assign severity levels to bug reports from various projects that are managed by Bugzilla bug tracking systems (Lamkanfi et al. 2010). They first extract word tokens and process them. These tokens are then fed into a Naive Bayes classifier to automatically assign a severity level to the corresponding bug. Different from the work by Menzies and Marcus, they classify coarse grained bug severity labels: severe, and non-severe. Three of the six classes of severity in Bugzilla (blocker, critical, and major) are grouped as severe, two of the six classes (minor, and trivial) are grouped as non- severe, and normal severity bugs are omitted from their analysis. Extending the above work, Lamkanfi et al. also explore various classification algorithms (Lamkanfi et al. 2011). They show that Naive Bayes classifier performs better than other approaches on a dataset of 29,204 bug reports.

Tian et al. propose an information retrieval based nearest neighbor approach to automatically assign bug report severity levels (Tian et al. 2012a). Similar to the work by Lamkanfi et al. they consider bug reports on Bugzilla repositories of various open source projects. They compare their approach with that of Menzies and Marcus on a dataset containing more than 65,000 bug reports and show that their approach could gain significant F-measure improvements.

Our study complements the aforementioned prior research efforts. In particular, we highlight that the results of prior efforts should be re-examined given the noisy nature of bug reports. Moreover we propose a new approach to assess the performance of automated classification approaches. Our proposed approach factors in the noisy nature of the ground truth that is commonly used by research efforts in this active area of research.

8.2 Noise and Inconsistency in Software Engineering Data

Software repositories contain a wealth of data related to a software system. However, the data is often not clean and contains noise or inconsistency in it (Mockus 2008; Zeller 2013). Various researchers have shown that noise in software repositories could impact research approaches, e.g., bug prediction (Herzig et al. 2013; Mockus 2008; Strike et al. 2001; Zeller 2013), that depend on such repositories of data.

Antoniol et al. showed that most of the issue reports reported as bug reports were not actually bug reports but other maintenance tasks, e.g., enhancement (Antoniol et al. 2008). They used DT, NBM, and logistic regression models to automatically distinguish bug reports from other kinds of maintenance tasks. Herzig et al. manually examined more than 7,000 issue reports of five different open-source projects (Herzig et al. 2013). Their results show that on average 39 % of the files marked as bug reports were not actually bug reports.

Ramler and Himmelbauer examined various causes of noise encountered during the defect prediction process (Ramler and Himmelbauer 2013). They conducted an experiment

[3]The tf-idf is commonly used to reflect the importance of a word to a document in a collection of documents. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the collection of documents.

to analyze the impact of noise on defect prediction performance. Their results show that the performance remains stable even at the 20 % noise rate. Zhu and Wu described a quantitative study of the impact of noise in machine learning techniques (Zhu and Wu 2004). Their results showed that noise in class labels can impact the accuracies of machine learning techniques.

Nguyen et al. showed that 11 %-39 % of the fixing commits are not purely corrective maintenance commits (Nguyen et al. 2013). In 3 %-41 % of mixed-purpose fixing commits, developers did not mention the type of maintenance activities that they performed. Nguyen et al. showed that approaches , that rely on historical information of fixed/buggy files, are affected by noise in the data, i.e., mixed-purpose fixing commits.

Kim et al. proposed approaches to deal with noise in defect prediction data (Kim et al. 2011). They found that adding false positive or false negative noises alone does not impact much the performance of two defect prediction models. However, a 20 %-30 % noise rate, i.e., including both false negatives and false positives, significantly decreased the performance of defect prediction models.

Nuseibeh et al. studied inconsistencies among software artifacts which are detected by checking for violations of some rules (Nuseibeh et al. 2000). They propose a framework to manage inconsistencies by diagnosing and handling them; the handling process could involve ignoring, deferring, circumventing, ameliorating, and resolving the inconsistencies. In this work, we also investigate inconsistencies, focusing on the values of the severity fields of bug reports, and propose a new evaluation metric to deal with such inconsistencies. In a future work, we plan to develop additional solutions to diagnose and handle inconsistencies on bug reports.

Kocaguneli et al. proposed a new software effort estimator called TEAK, that runs a clustering algorithm over effort data to find, then remove, regions of high effort variance. Estimation is then performed on the surviving data; the surviving data is hierarchically clustered and estimator is generated from this tree by using a recursive descend algorithm that terminates when the variance of a sub-tree is higher than that of its super-tree (Kocaguneli et al. 2012).

Our study shows that noise exists in another type of software engineering data, i.e., bug severity data. We also present a novel approach that future research efforts should use to assess their proposed approaches given the noisy nature of bug severity data.

9 Conclusion

We observed that up-to 51 % of the bug severity labels are unreliable. The accuracy of classifiers trained on unreliable data suffers as the unreliability of the data increases. While our results do indicate that duplicate bug reports have unreliable severity labels, we believe that they send warning signals about the reliability of the full bug severity data (i.e., including non-duplicate reports). More empirical studies should be conducted to explore if our findings generalize to the full dataset. Our work addresses an important consideration that has not been addressed by many past studies on bug severity prediction (Menzies and Marcus 2008; Lamkanfi et al. 2010, 2011; Tian et al. 2012a).

Our results should not deter others from using such data; however, careful care should be used when working with such unreliable data and close attention should be observed when interpreting results derived from such unreliable data.

We also proposed a new approach to better assess the accuracy of automated classifiers of severity labels given the unreliable nature of the data. Our assessment approach shows

that current automated approaches perform well – 77-86 % agreement with human-assigned severity labels.

In the future, we plan to use duplicate bug reports to check for consistency of other bug report data (e.g., component, priority, etc.). We are also interested to propose a technique to identify and better manage inconsistent bug report data.

References

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: A text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds, CASCON '08,. ACM, New York, pp 23:304–23:318

Cohen WW (1995) Fast effective rule induction. In: Proceedings of the 12th international conference on machine learning (ICML'95), pp 115–123

Espinha T, Zaidman A, Gross HG (2014) Web api growing pains: stories from client developers and their code. In: Software evolution week-IEEE conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE), 2014. IEEE, pp 84–93

Hayes AF, Krippendorff K (2007) Answering the call for a standard reliability measure for coding data. Commun Method Meas 1(1):77–89

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 392–401

Huang SJ, Lin CY, Chiu NH et al (2006) Fuzzy decision tree approach for embedding risk assessment information into software cost estimation model. J Inf Sci Eng 22(2):297–313

Joachims T (1998) Text categorization with suport vector machines: learning with many relevant features. In: Proceedings of the 10th European conference on machine learning, ECML '98. Springer-Verlag, London, pp 137–142

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceedings of the 33rd international conference on software engineering, ICSE '11. ACM, New York, pp 481–490

Krippendorff K (2003) Content analysis: an introduction to its methodology, 2nd edn. Sage Publications

Kocaguneli E, Menzies T, Bener AB, Keung JW (2012) Exploiting the essential assumptions of analogy-based effort estimation. IEEE Trans Software Eng 38(2):425–438

Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: 7th IEEE working conference on mining software repositories (MSR), 2010. IEEE, pp 1–10

Lamkanfi A, Demeyer S, Soetens QD, Verdonck T (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: 15th European conference on software maintenance and reengineering (CSMR), 2011, pp 249–258

Lim H, Goel A (2006) Support vector machines for data modeling with software engineering applications. In: Pham H (ed) Springer handbook of engineering statistics. Springer, London, pp 1023–1037

Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: IEEE international conference on software maintenance, 2008. ICSM 2008. IEEE, pp 346–355

Mockus A (2008) Missing data in software engineering. In: Shull F, Singer J, Sjberg D (eds) Guide to advanced empirical software engineering. Springer, London, pp 185–200. doi:10.1007/978-1-84800-044-5_7

Nguyen A, Nguyen T, Nguyen T, Lo D, Sun C (2012) Duplicate bug report detection with a combination of information retrieval and topic modeling. In: ASE. ACM, pp 70–79

Nguyen HA, Nguyen AT, Nguyen T (2013) Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization

Nuseibeh B, Easterbrook S, Russo A (2000) Leveraging inconsistency in software development. Computer 33(4):24–29

Ramler R, Himmelbauer J (2013) Noise in bug report data and the impact on defect prediction results. In: Joint conference of the 23rd international workshop on software measurement and the 2013 8th international conference on software process and product measurement (IWSM-MENSURA), 2013, pp 173–180

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: ICSE. IEEE, pp 499–510

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Ki Matsumoto (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042

Strike K, El Emam K, Madhavji N (2001) Software cost estimation with incomplete data. IEEE Trans Softw Eng 27(10):890–908
Sun C, Lo D, Wang X, Jiang J, Khoo S (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: ICSE. ACM, pp 45–54
Thung F, Lo D, Jiang L, Rahman F, Devanbu PT et al (2012) When would this bug get reported? In: 28th IEEE International Conference on Software Maintenance (ICSM), 2012. IEEE, pp 420–429
Tian Y, Lo D, Sun C (2012a) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 19th working conference on reverse engineering (WCRE), 2012. IEEE, pp 215–224
Tian Y, Sun C, Lo D (2012b) Improved duplicate bug report identification. In: 16th European conference on software maintenance and reengineering, CSMR 2012, Szeged, pp 385–390
Tian Y, Lo D, Xia X, Sun C (2014) Automated prediction of bug report priority using multi-factor analysis. Empir Softw Eng:1–30
Valdivia Garcia H, Shihab E (2014) Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 72–81
Vyas D, Fritz T, Shepherd D (2014) Bug reproduction: A collaborative practice within software maintenance activities. In: COOP 2014-Proceedings of the 11th international conference on the design of cooperative systems. Springer, Nice (France), pp 189–207
Xia X, Lo D, Wen M, Shihab E, Zhou B (2014) An empirical study of bug report field reassignment. In: 2014 software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering, CSMR-WCRE 2014. Antwerp, Belgium, pp 174–183
Zeller A (2013) Can we trust software repositories? In: Mnch J, Schmid K (eds) Perspectives on the future of software engineering. Springer, Berlin Heidelberg, pp 209–215
Zhang F, Khomh F, Zou Y, Hassan AE (2012) An empirical study on factors impacting bug fixing time. In: 19th Working Conference on Reverse Engineering (WCRE), 2012. IEEE, pp 225–234
Zhang H, Gong L, Versteeg S (2013) Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 1042–1051
Zhu X, Wu X (2004) Class noise vs. attribute noise: a quantitative study of their impacts. Artif Intell Rev 22(3):177–210

Yuan Tian is currently a PhD student in the School of Information Systems, Singapore Management University. She started her PhD program in 2012. Previously, she received her bachelor degree in the College of Computer Science and Technology from Zhejiang University, China in 2012. Her research is in software system and data mining area. Particularly, she is interested in analyzing textual information in software repositories.

Nasir Ali is a Postdoctoral fellow at the Department of Electrical and Computer Engineering of University of Waterloo, working with Prof. Lin Tan. He has received his Ph.D. at Ecole polytechnique de Montreal under the supervision of Prof. Yann-Gaël Guéhéneuc and Prof. Giuliano Antoniol. The primary focus his Ph.D. thesis was to develop tools and techniques to improve the quality of software artifacts' traceability. He received a M.Sc. in computer science and an M.B.A. from the University of Lahore and National College of Business Administration & Economics, respectively. He has more than six years of industrial experience. His research interests include software maintenance and evolution, system comprehension, and empirical software engineering.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 130 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009. He has won a number of research awards including an ACM distinguished paper award for his work on bug report management. He has published in many top international conferences in software engineering, programming languages, data mining and databases, including ICSE, FSE, ASE, PLDI, KDD, WSDM, TKDE, ICDE, and VLDB. He has also served on the program committees of ICSE, ASE, KDD, VLDB, and many others. He is a steering committee member of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) which is a merger of the two major conferences in software engineering, namely CSMR and WCRE. He will also serve as the general chair of ASE 2016. He is a leading researcher in the emerging field of software analytics and has been invited to give keynote speeches and lectures on the topic in many venues, such as the 2010 Workshop on Mining Unstructured Data, the 2013 Gnie Logiciel Empirique Workshop, the 2014 International Summer School on Leading Edge Software Engineering, and the 2014 Estonian Summer School in Computer and Systems Science.

Ahmed E. Hassan is a Canada Research Chair in Software Analytics and the NSERC/Blackberry Industrial Research Chair at the School of Computing in Queen's University. Dr. Hassan serves on the editorial board of the IEEE Transactions on Software Engineering, the Journal of Empirical Software Engineering, and PeerJ Computer Science. He spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community.

Early tools and techniques developed by Dr. Hassan's team are already integrated into products used by millions of users worldwide. Dr. Hassan industrial experience includes helping architect the Blackberry wireless platform, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Dr. Hassan is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada, and Japan. More information at: http://sail.cs.queensu.ca/.