Automatic Fine-Grained Issue Report Reclassification

Pavneet Singh Kochhar, Ferdian Thung, and David Lo
School of Information Systems
Singapore Management University, Singapore
{kochharps.2012, ferdiant.2013, davidlo}@smu.edu.sg

***Abstract*—Issue tracking systems are valuable resources during software maintenance activities. These systems contain different categories of issue reports such as bug, request for improvement (RFE), documentation, refactoring, task etc. While logging issue reports into a tracking system, reporters can indicate the category of the reports. Herzig et al. recently reported that more than 40% of issue reports are given wrong categories in issue tracking systems. Among issue reports that are marked as bugs, more than 30% of them are not bug reports. The misclassification of issue reports can adversely affects developers as they then need to manually identify the categories of various issue reports. To address this problem, in this paper we propose an automated technique that reclassifies an issue report into an appropriate category. Our approach extracts various feature values from a bug report and predicts if a bug report needs to be reclassified and its reclassified category. We have evaluated our approach to reclassify more than 7,000 bug reports from HTTPClient, Jackrabbit, Lucene-Java, Rhino, and Tomcat5 into 1 out of 13 categories. Our experiments show that we can achieve a weighted precision, recall, and F1 (F-measure) score in the ranges of 0.58-0.71, 0.61-0.72, and 0.57-0.71 respectively. In terms of F1, which is the harmonic mean of precision and recall, our approach can substantially outperform several baselines by 28.88%-416.66%.**

Keywords*—*Issue Reports, Fine-Grained, Reclassification

I. Introduction

Issue tracking system (IST), which contains information related to issues faced during the development as well as after the release of a software project, is an integral part of software development activity. Using a tracking system, such as a JIRA or Bugzilla system, issue reporters can submit various kinds of reports, including, bug reports, documentations, feature requests, refactoring request, and many more. The number of these issue reports however could be too large for developers to handle. Anvik et al. quoted a Mozilla developer who commented that "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [1]. This highlights the need for solutions that can help developers to cope with the mass of issue reports that are submitted to ISTs.

One problem that developers face is the incorrect assignment of fields in bug reports. Xia et al. find that developers require more time to fix issue reports whose fields are wrongly assigned than those that are assigned correctly [2]. An important piece of information in an issue report is the category of the report, which describes if the report is about a bug, adaptive maintenance, perfective maintenance, documentation update, test case creation, etc. These categories can be inferred from the type field in JIRA and importance field in Bugzilla. Based on these categories, developers can prioritize reports and group related reports together. Herzig et al. study the categories of more than 7,000 bugs across 5 software systems and find that more than 40% of issue reports are given wrong categories [3]. Among issue reports assigned as `BUG`, more than 30% of them are not bug reports. The misclassification of issue reports can adversely affect developers as they then need to manually identify the categories of various issue reports. Due to the overwhelming number of issues reported in issue tracking systems, developers have to spend substantial amount of time to manually analyze issue reports and assign them the correct categories. This manual process is tedious and becomes increasingly difficult as the project's user base grows as there would then be more users reporting issues.

To address the above-mentioned problem, there is a need for an approach that can automatically reclassify issue reports to their actual fine-grained categories. We refer to the corresponding problem as *fine-grained issue report reclassification*. In this work, we fill this need by proposing an approach that takes in an issue report and outputs whether the issue report needs to be reclassified or not and the reclassified category. Our tool extracts a number of features from an issue report including the words that appear in the summary and description fields of a bug report, the bug reporter identifier, the presence of stack trace, and the original category given by the reporter. Next, based on a training set of issue reports, we build a statistical model that can be used to predict if the category of a new issue report needs to be changed and its reclassified category.

Antoniol et al. proposed an approach that can classify an issue report as bugs or enhancements [4]. Our work is different from Antoniol et al.'s work in two ways: first, different from Antoniol et al. that only consider two categories (bug reports vs. enhancements), we consider *13 categories* based on the categories used by Herzig et al. in their manual reclassification effort [3][1]; second, we consider a *reclassification* problem, and thus we do not consider the reported category to be the ground truth, rather we use the reported category as a feature to build a statistical model.

We have evaluated our proposed approach on more than 7,000 bug reports from five software systems: HTTPClient,

[1]In their paper, 6 categories are identified. We use the categories in their publicly downloadable dataset which include 14 categories. We merge `UNKNOWN` to `OTHERS`.

Jackrabbit, Lucene-Java, Rhino, and Tomcat5. These bug reports have previously been manually analyzed by Herzig et al. which reclassified the categories of more than 40% of them [3]. We use Herzig et al.'s manually assigned categories as ground truth to evaluate our proposed automated reclassification approach. The experiment results show that across the five software systems, we can achieve precision, recall, and F-measure scores of up to 0.71, 0.72, and 0.71 respectively. We have also compared our approach to a number of baseline approaches. We show that we can outperform these baseline approaches by 28.88%-414.66%.

The contributions of this paper are as follows:

1) We introduce the problem of *fine-grained issue report reclassification* which extends the report classification problem first proposed by Antoniol et al. [4]. The goal of issue report reclassification is to predict issue reports with wrong categories and assign to them fine-grained new categories.
2) We propose an approach that extracts textual, author, and stack trace feature values, and use them along with the reported categories to build a statistical model that can predict a fine-grained category that should be assigned to an issue report. We have evaluated our approach on more than 7,000 issue reports from five systems and compare it with several baselines. Our experiments show that our proposed approach can outperform the baselines by a substantial margin.

The structure of this paper is as follows. In Section II, we describe preliminary information on issue reports, text pre-processing and classification algorithms that we use in this study. In Section III, we present our proposed approach which takes as input a bug report and predicts if its category is wrong and if so, recommends a new category. In Section IV, we present our experimental results. We discuss related work in Section V. We finally conclude and mention future work in Section VI.

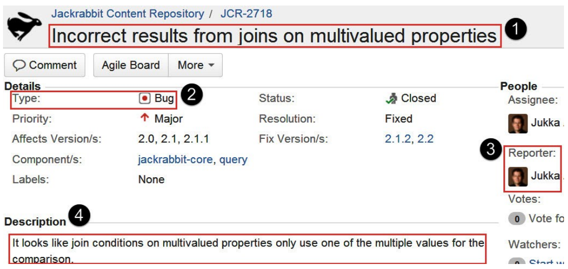
II. Preliminaries

In this section, we discuss some preliminary information about issue reports, text-preprocessing, and classification algorithms that we use in this work.

A. Issue Reports

Figure 1 shows an issue report from the JIRA system of Jackrabbit project. Notice that an issue report contains several fields that carry several pieces of information. In this work, we are particularly interested in the following: short summary (1), issue category (2), reporter (3), and longer description (4).

Herzig et al. have manually labeled issue reports and they assign them to 13 categories: `BUG`, `RFE`, `IMPROVEMENT`, `DOCUMENTATION`, `TASK`, `BUILD SYSTEM`, `REFACTORING`, `DESIGN DEFECT`, `TEST`, `CLEANUP`, `BACKPORT`, `SPECIFICATION`, and `OTHERS`. We describe the meaning of each of these categories in Table I.


Jackrabbit Content Repository / JCR-2718

Incorrect results from joins on multivalued properties 1

Details

Type:	Bug 2	Status:	Closed
Priority:	Major	Resolution:	Fixed
Affects Version/s:	2.0, 2.1, 2.1.1	Fix Version/s:	2.1.2, 2.2
Component/s:	jackrabbit-core, query		
Labels:	None		

People

Assignee: Jukka

3 Reporter: Jukka

Votes: 0 Vote fo

Watchers:

Description 4

It looks like join conditions on multivalued properties only use one of the multiple values for the comparison.

Figure 1: Example Issue Report of Jackrabbit Project with BugID JCR-2718.

B. Text Pre-Processing

The summary and description field of bug reports contain textual contents. Several pre-processing strategies are typically performed on textual contents. These include: stop-word removal and stemming. After stop-word removal and stemming process, we represent the summary and description field of each bug report as a bag (i.e., a multiset) of words.

Stop-word removal refers to the removal of commonly used words that carry little meaning. These words include "is", "are", "am", "if", etc. To remove stop words, we consider a list of stop words that are provided at http://dev.mysql.com/doc/refman/5.6/en/fulltext-stopwords.html. Stemming refers to the process of reducing a word to its root form. For example, the word "reads" and "reading" would both be reduced to "read". By doing stemming, words that carry very similar meaning would be mapped to the same token. We make use of Porter stemmer, which removes the suffixes of words to reduce a word to its root form [5]. We use the implementation of Porter Stemmer available from: http://tartarus.org/martin/PorterStemmer/.

C. Classification Algorithms

Here, we describe each of the classification algorithms that is used in this study.

1) **Support Vector Machine**
Support Vector Machine (SVM) [6] constructs a hyperplane or a set of hyperplanes in n-dimensional space. Each training bug report is represented as a point in a multi-dimensional space where each feature represents a dimension. SVM selects a small number of critical boundary instances as support vectors for each class, and builds a discriminant function to form decision boundaries with the principle of maximizing the margins among training issue reports belonging to the different labels. The discriminant function itself can be tuned based on the kernel that is used. The most commonly used one is linear kernel.

2) **Naive Bayes**
Naive Bayes [7] works under the assumption that each feature is independent to other features in discriminating the class label (in our case: the 13 issue report categories). Based on this assumption, for an instance

Table I: Thirteen Issue Categories Manually Assigned By Herzig et al. to Issue Reports

Category	**Description**
BUG	Issue reports documenting a problem which impairs or prevents the correct functioning of a software and causes deviation from expected results. A bug can be an error, defect, failure or fault.
RFE	Issue reports which document request for enhancement (RFE) such as addition of new functionality or a new feature.
IMPROVEMENT	Issue reports specifying perfective maintenance task to improve the overall performance of a software, e.g., to change a piece of code to produce results from a database faster.
DOCUMENTATION	Issue reports which refer to updating external links or documentation related to code, e.g., to update API documentation in HTML format.
TASK	Issue reports which specify a task that needs to be done.
BUILD SYSTEM	Issue reports related to problems in build systems, which are used to automate several software build activities such as compiling code and running test cases.
REFACTORING	Issue reports specifying refactoring of source code, i.e., changing the non-functional attributes of a piece of code which improves its maintainability.
DESIGN DEFECT	Issue reports pertaining to problems in the design of a software, e.g., code smells.
TEST	Issue reports which are related to test cases.
CLEANUP	Issue reports pertaining to code cleanup, e.g., to clean unwanted code such as redundant code, dead code, etc.
BACKPORT	Issue reports related to backporting where a fix or patch of any flaw on the current version is applied to an older version of a system.
SPECIFICATION	Issue reports related to changes in the requirement specification documents.
OTHERS	Issue reports that cannot be classified to any of the above categories.

$I_i = \langle f_1, f_2, ..., f_n \rangle$, where I_i is i-th data instance, f_j is the j-th feature that is extracted to classify I_i, n is the number of features, and m is the number of classes, we can compute the probability that an instance I_i is categorized as class C_i by the following formula.

$$\begin{aligned} p(C_i = c|I_i) &= \frac{p(C_i = c) \times p(I_i|C_i = c)}{\sum_{c' \in \{c_1,...,c_m\}} p(C_i = c') \times p(I_i|C_i = c')} \\ &= \frac{p(C_i = c) \times \prod_{i=1}^{|I_i|} p(f_i|C_i = c)}{\sum_{c' \in \{c_1,...,c_m\}} p(C_i = c') \times \prod_{i=1}^{|I_i|} p(f_i|C_i = c')} \end{aligned}$$

The above equation can be used to predict a class label for an instance I_i. For example, if $\forall_{k \in c' \setminus \{c\}}$ $p(C_i = c|I_i) \geq p(C_i = k|I_i)$, then we classify I_i as class c. Due to the assumed independence of the features, naive Bayes is computationally cheap compared to other algorithms that assume no such independence. Despite the inaccuracy of the assumption in many real world problems, it has been shown to work pretty well.

3) **Naive Bayes Multinomial**
Naive Bayes Multinomial [7] is an extension of the Naive Bayes algorithm. Notice that in Naive Bayes, it only considers the presence or absence of a feature in an instance, but not the actual value of the feature. Naive Bayes Multinomial considers the actual value. In general, Naive Bayes Multinomial performs better than Naive Bayes when the total number of unique features are large. It has been successfully used in many past studies, e.g., [8].

4) **K-Nearest Neighbor**
K-nearest neighbor (kNN) [6] is an instance-based algorithm based on the idea that a particular class instances should be close to one another. Thus, to predict the label of an instance, kNN perform two steps:
   a) For each unlabeled instance, kNN searches for its first k nearest neighbors in the labeled instances. Distance between two data instances is measured using a particular distance metric (e.g., Euclidean distance, Minkowsky distance, Manhattan distance, etc).
   b) kNN then assigns to each unlabeled instance the most frequent label that its k-nearest neighbors have. For example, if the number of neighboring labeled instances having class c is higher than those having other class labels, kNN classifies it as class c.

5) **Random Forest**
Random Forest [9] is an extension of classification tree. Classification tree is constructed by iteratively picking a feature that can "best" separate the instances to different classes. This is usually measured by an attribute selection metric, e.g., information gain. The process continues until the leafs of the tree contains only instances with the same class label. Instead of making one classification tree by considering the entire features in the data, random forest builds many classification trees; each corresponding to a randomly chosen subset of the entire features. To classify a new instance, the feature vector of the instance is input to each of the classification tree. Each tree produces a particular classification. This is often called the tree "vote" for the class. The algorithm then chooses the classification having the most votes.

6) **RBF Network**
RBF Network [10] is a variant of artificial neural network whose activation function in the hidden layer of the network is a radial basis function. It outputs a score that is computed by linear combination of input, network

parameter, and radial basis function. During training, each class label is assigned a specific score. Given a labeled instance, RBF Network then trains the network to output a score as close as possible to the labeled score. To classify an unlabeled instance, RBF Network picks the class label whose score is the closest to the instance score.

III. Proposed Approach

In this section, we first present the overall framework of our proposed approach. We then zoom-in to the feature extraction component to describe the set of features that we use, and the model building component to describe how we convert vectors of feature values to a statistical model.

A. Overall Framework

Our framework, shown in Figure 2, is divided into two phases: training and deployment. The goal of the training phase is to create a statistical model to predict whether an issue report needs to be reclassified and its reclassified category. In the deployment phase, we use the model created during the training phase to predict the reclassified category of a new issue report.

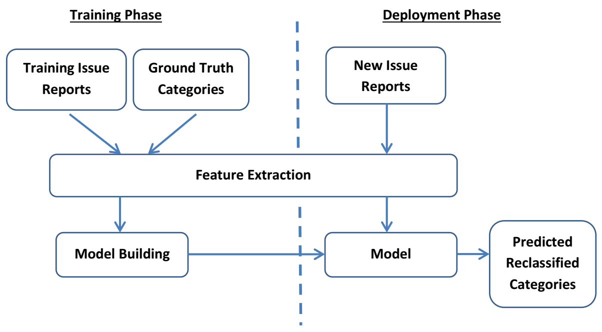


Figure 2: Overall Framework

To train the statistical model, in the training phase, our approach takes as input a training set of issue reports along with their ground truth categories. These ground truth categories can be obtained by manual inspection. Herzig et al. has a good dataset that contains ground truth categories of more than 7,000 issue reports [3]. Based on these inputs, our approach then extracts values of various features from each issue report. Each feature corresponds to a characteristic of an issue report. At the end of this step each issue report is represented by a vector of feature values. These vectors are then input to the model builder component which employs a machine learning solution to build a statistical model that is able to predict the reclassified category of an issue report.

After the model is built, it can then be used in the deployment phase to assign reclassified category to a new issue report. To do so, we first extract values of various features from the new issue report and create a representative feature vector. This vector is then input to the model which outputs the reclassified category.

In the next subsections, we elaborate our feature extraction and model building processes.

B. Feature Extraction

We extract a number of features from an issue report which include the preprocessed words extracted from its summary and description fields as well as other information including: the reported category of the report, the issue reporter identifier, and the presence of a stack trace in the description field. Table II describes the features that we extract in this study.

Each preprocessed word in the summary and description fields is a feature and its value is the weight of the word computed using the standard TF-IDF weighting scheme [11]. Using this weighting scheme, the weight of a word is the product of its term frequency (TF) and inverse document frequency (IDF). Term frequency of a word corresponds to the number of times the word appear in an issue report. Inverse document frequency of a word corresponds to the logarithm of the ratio of the total number of issue reports to the number of issue reports in which this word occurs. Higher value of IDF shows that the term is rare and can be used to differentiate between issue reports, whereas low value shows that the term is very common. The weight of a word in an issue report can then be computed as follows:

$$w_{i,D,C} = TF_{i,D} \times IDF_{i,C}$$
$$IDF_{i,C} = log(\frac{N}{n_i})$$

In the above equations, $TF_{i,D}$ refers to the number of times a word i appears in an issue report D. N is the total number of issue reports and n_i is the number of issue reports which contain the word i.

For the stack trace feature (F_2), we detect the existence of a stack trace in the description field of an issue report by checking the occurrence of one of the following patterns:

1) Phrase: "Exception in thread"

2) Regular expression:
[A-Za-z0-9$.]+Exception

3) Regular expression:
[A-Za-z0-9$.]+[A-Za-z0-9]+([A-Za-z0-9]+(java:[0-9]+)?)

These patterns often appear in a stack trace of programs written in Java. The five projects that we analyze in this study are all written in Java.

C. Model Building

At the end of the feature extraction process, each issue report is mapped to a vector of feature values (aka. a feature vector). In the model building process, our approach takes as input a set of feature vectors corresponding to training bug reports and produces a statistical model. The model building process uses a classification algorithm to build this model.

Table II: List of Features

ID	Feature
W_1 - W_N	Each feature is a preprocessed word that appears in the summary or description fields of an issue report. Here, N is the total number of words.
C_1 - C_{13}	Reported category of an issue report as indicated in an issue tracking system. For example, if the issue report is marked as a *bug* by its reporter, the value of that feature is 1. This feature vector would have one of the features as 1 and rest of the features as 0 (Table I).
S	Boolean feature that indicates the presence or absence of a stack trace in the description field of a bug report. Its value is 1 if a stack trace exists and 0 otherwise. We use a number of regular expressions to automatically identify the presence or absence of a stack trace.
R_1 - R_M	The identifier of the reporter of the issue report. Each reporter is treated as a feature in one-dimension of the n-dimensional feature vector. The value of that feature is 1 if the reporter reported that bug. Here, M is the total number of reporters.

We need a classification algorithm that is able to build a model that can classify a data instance (i.e., an issue report) into one out of 13 classes (i.e., the 13 categories). A classification algorithm that is able to perform this task is often known as a multi-class classification algorithm. There are a number of such algorithms (some of them are presented in Section II-C), in this paper, by default we make use of an implementation of Support Vector Machine (SVM) named LibSVM [12][2] which has been shown to be effective in many prior studies, e.g., [13].

IV. Experiment

In this section, we first describe the datasets that we use and our experiment methodology. Next we present our research questions and our experiment results that answer these questions.

A. Datasets & Experiment Methodology

We use the manually reclassified issue report datasets from Herzig et al. [3], which is made publicly available from http://www.st.cs.uni-saarland.de/softevo//bugclassify/.
The dataset contains a total of 7401 closed, resolved, and verified issue reports from 5 open source Java projects. Table III describes this dataset.

Table III: Project Description

Project	Organisation	Tracker Type	# of Bug Reports
HTTPClient	APACHE	JIRA	746
Jackrabbit	APACHE	JIRA	2402
Lucene-Java	APACHE	JIRA	2443
Rhino	MOZILLA	BugZilla	1226
Tomcat5	APACHE	BugZilla	584

To measure the effectiveness of our approach, we perform a 10-fold stratified cross validation [14]. We divide the dataset into 10 buckets, where each bucket contains similar proportion of issue reports of various ground truth categories. We then use nine of these buckets as training data, and one as test data. We evaluate the performance of a issue report reclassification approach on the test data. We repeat the process ten times and compute the average performance. We measure the performance of an issue report reclassification approach in terms of weighted precision, recall, and F-measure. We define these three metrics as follows:

$$\begin{aligned}
Prec_{category} &= \frac{\#TP_{category}}{\#TP_{category}+\#FP_{category}} \\
Rec_{category} &= \frac{\#TP_{category}}{\#TP_{category}+\#FN_{category}} \\
F1_{category} &= \frac{2\times Prec_{category}\times Rec_{category}}{Prec_{category}+Rec_{category}} \\
WPrec &= \frac{1}{N}\times\sum_{category=1}^{\#category} n_{category}\times Prec_{category} \\
WRec &= \frac{1}{N}\times\sum_{category=1}^{\#category} n_{category}\times Rec_{category} \\
WF1 &= \frac{1}{N}\times\sum_{category=1}^{\#category} n_{category}\times F1_{category}
\end{aligned}$$

In the above equations, $\#TP_{category}$, $\#FP_{category}$, and $\#FN_{category}$ are the number of true positives, false positives, and false negatives of a particular category. N is the total number of issue reports in the test data, and $n_{category}$ is the number of issue reports of a particular category.

We compare our approach with the two baseline approaches. The first baseline (Baseline-1) simply predicts the reclassified category of an issue report to be the same as its original category. For example, for the issue report in Figure 1, the first baseline approach would predict its reclassified category as "Bug". The second baseline (Baseline-2) simply predicts the reclassified category of every issue report as "Bug". Note that "Bug" is the most common category among all issue reports.

B. Research Questions

We investigate 5 research questions in our study:

RQ1: To what extent could our proposed approach predict if an issue report needs to reclassified, and if so, outputs its correct category? We use our proposed approach to predict the correct category of an issue report and compare this category with the ground truth which is manually created by Herzig et al. We compute the weighted precision, recall and F-measure of our approach and compare them with the

[2]Available for download from: http://www.csie.ntu.edu.tw/~cjlin/libsvm/

Table IV: Effectiveness of Our Approach. Prec = Precision, Rec = Recall, F1 = F-Measure.

	HTTPClient			**Jackrabbit**			**Lucene-Java**		
% of Issue Reports	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
Ours	0.61	0.63	0.60	0.71	0.72	0.71	0.62	0.63	0.62
Baseline-1	0.54	0.52	0.43	0.61	0.62	0.54	0.50	0.50	0.43
Baseline-2	0.16	0.40	0.23	0.15	0.39	0.21	0.08	0.28	0.12
Improvement-1 (%)	12.96	21.15	**39.53**	16.39	16.12	**31.48**	24.00	26.00	**44.18**
Improvement-2 (%)	281.25	57.49	**160.86**	373.33	84.61	**238.09**	675.0	125.0	**416.66**

	Rhino			**Tomcat5**		
% of Issue Reports	Prec	Rec	F1	Prec	Rec	F1
Ours	0.58	0.61	0.57	0.58	0.62	0.58
Baseline-1	0.35	0.57	0.43	0.36	0.58	0.45
Baseline-2	0.26	0.51	0.35	0.30	0.54	0.38
Improvement-1 (%)	65.71	7.01	**32.55**	61.11	6.89	**28.88**
Improvement-2 (%)	123.07	19.60	**62.85**	93.33	14.81	**52.63**

baseline approaches.

RQ2: What is the effect of varying the amount of training data on the effectiveness of our proposed approach? In RQ1, we use ten-fold cross validation, which means that 90% of the labeled issue reports are used as training data. In this research questions, we would like to investigate the impact of reducing the amount of training data on the performance of our approach.

RQ3: What are the features that can discriminate between different issue categories? We extract thousands of features from issue reports. Here, we analyse and identify features which are good at differentiating between different categories. We use Fisher score [15] to infer the most discriminative features. The Fisher score for a feature is computed as:

$$F(j) = \frac{\sum_{class=1}^{\#class}(\overline{x}_j^{(class)} - \overline{x}_j)^2}{\sum_{class=1}^{\#class}(\frac{1}{n_{class}-1}\sum_{i=1}^{n_{class}}(x_{i,j}^{(class)} - \overline{x}_j^{(class)})^2)}$$

In the above equation, $F(j)$ represents the Fisher score of the j^{th} feature, n_{class} is the numbers of issue reports of a particular category, $\overline{x_j}$ represents the average value of the j^{th} feature of all issue reports, $\overline{x}_j^{(class)}$ is the average value of the j^{th} feature of the i^{th} category-labeled issue report and $x_{i,j}^{(class)}$ is the j^{th} feature of the i^{th} category-labeled issue report.

RQ4: What kinds of issue reports are often misclassified by our proposed approach? In this question, we analyse the issue reports which are classified correctly and those which are misclassified by our proposed approach. We highlight the weaknesses of our approach.

RQ5: What are the effectiveness of various common classification algorithms for fine-grained issue report reclassification problem? By default we use a Support Vector Machine (i.e., LibSVM) as the classification algorithm. In this RQ, we compare the results of different classification algorithms with those of LibSVM.

C. Experiment Results

In this subsection, we present our experiment results which answer the five research questions presented earlier. We present an answer to each of these questions one at a time in the following subsections.

1) RQ1: Effectiveness of Our Approach: Table IV shows the Precision (Prec), Recall (Rec) and F-measure (F1) scores for our method and the two baselines. The results show that our approach performs better than both the baseline approaches for all the 5 projects. We can achieve a weighted precision, recall, and F-measure scores in the ranges of 0.58-0.71, 0.61-0.72, and 0.58-0.71 respectively. In terms of F1, which is the harmonic mean of precision and recall, our approach can substantially outperform the baselines by 28.88%-416.66%. The values in bold show the improvement of our approach over the two baselines.

2) RQ2: Varying the Amount of Training Data: We use different amount of training data ranging from 10% to 90 % of all issue reports in a dataset. For each percentage level, we randomly sample an issue report dataset 10 times to create training and test data sets. We report the average performance across the 10 iterations.

Table V shows the result of varying the amount of training data to build a model. We can observe that for HTTPClient, the F-measure remains stable when the amount of training data is varied from 50% to 90%. However there is a substantial reduction in F-measure when the amount of training data is reduced to 40% or lower. For Jackrabbit, the F-measure remains stable when the amount of training data is varied from 30% to 90%. However there is a substantial reduction in F-measure when the amount of training data is reduced to 20% or lower. For Lucene, the F-measure remains stable when the amount of training data is varied from 50% to 90%. However there is a substantial reduction in F-measure when the amount of training data is reduced to 40% or lower. For Rhino, the F-measure remains stable when the amount of training data is varied from 60% to 90%. However there is a substantial reduction in F-measure when the amount of training data is reduced to 50% or lower. For Tomcat5, the F-measure remains stable when the amount of training data is varied from 30% to 90% (with the exception of 70%). However there is a substantial reduction in F-measure when the amount of training data is reduced to 20% or lower. The above result shows that our approach requires a substantial number of training data (i.e., 30-60% of all issue reports) for it to work optimally. Note that even on the worst setting (i.e., 10% of all issue reports), in general our approach still outperforms or has a

Table V: Varying the Amount of Training Data. Prec = Precision, Rec = Recall, F1 = F-measure.

	HTTPClient			**Jackrabbit**			**Lucene-Java**		
% of Issue Reports	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
10	0.49	0.56	0.47	0.63	0.65	0.60	0.55	0.57	0.53
20	0.54	0.55	0.46	0.64	0.665	0.61	0.57	0.57	0.54
30	0.58	0.60	0.54	0.68	0.70	0.67	0.59	0.60	0.58
40	0.54	0.53	0.48	0.69	0.71	0.68	0.59	0.58	0.56
50	0.58	0.61	0.57	0.69	0.71	0.69	0.62	0.63	0.61
60	0.59	0.62	0.58	0.64	0.65	0.62	0.61	0.62	0.61
70	0.60	0.62	0.58	0.70	0.72	0.70	0.62	0.63	0.62
80	0.62	0.68	0.61	0.70	0.72	0.70	0.63	0.64	0.63
90	0.61	0.64	0.60	0.71	0.73	0.71	0.62	0.63	0.62

	Rhino			**Tomcat5**		
% of Issue Reports	Prec	Rec	F1	Prec	Rec	F1
10	0.45	0.52	0.40	0.47	0.54	0.43
20	0.46	0.50	0.39	0.50	0.55	0.45
30	0.46	0.50	0.40	0.54	0.60	0.53
40	0.47	0.48	0.40	0.56	0.62	0.56
50	0.52	0.58	0.50	0.56	0.61	0.56
60	0.55	0.59	0.53	0.50	0.48	0.42
70	0.56	0.60	0.54	0.49	0.44	0.38
80	0.58	0.61	0.56	0.57	0.62	0.58
90	0.59	0.61	0.56	0.54	0.59	0.55

similar performance to the baseline approaches.

3) RQ3: Most Discriminative Features: We compute a list of top-10 most discriminative features for each project. Tables VI-X shows the list of top-10 features with their corresponding Fisher scores.

We note that many of the discriminative features correspond to the value of the reported category. Some words are also useful indicators to predict the correct category of an issue report. For example, the presence of the word "test" often indicates that an issue report is of category `TEST`, the presence of the word "cleanup" often indicates that an issue report is of category `CLEANUP`, the presence of the word "maven"[3] often indicates that an issue report is of category `BUILD SYSTEM`, the presence of the word "design" often indicates that an issue report is of category `DESIGN DEFECT`, and so on.

Table VI: Most Effective Features (HTTPClient)

No.	Feature	Fisher Score
1	Stemmed word "test"	1.73
2	Reported Category (`TASK`)	0.58
3	Stemmed word "privat"	0.56
4	Reported Category (`BUG`)	0.54
5	Stemmed word "cleanup"	0.50
6	Stemmed word "protect"	0.48
7	Stemmed word "maven"	0.48
8	Stemmed word "design"	0.43
9	Reported Category (`TEST`)	0.43
10	Stemmed word "remov"	0.39

4) RQ4: Analysis of Correctly & Wrongly Classified Issue Reports: We draw a confusion matrix to analyse issue reports that are correctly and wrongly classified by our proposed approach. Figure XI shows the confusion matrix. The x-axis corresponds to categories predicted by our approach.

[3]Maven is a utility that developers can use to build a software project.

Table VII: Most Effective Features (Jackrabbit)

No.	Feature	Fisher Score
1	Reported Category (`BUG`)	0.72
2	Stemmed word "test"	0.55
3	Stemmed word "maven"	0.51
4	Stemmed word "backport"	0.46
5	Reported Category (`IMPR`)	0.43
6	Stemmed word "remov"	0.37
7	Reported Category (`TASK`)	0.34
8	Stemmed word "branch"	0.34
9	Stemmed word "open"	0.33
10	Stemmed word "issu"	0.24

Table VIII: Most Effective Features (Lucene-Java)

No.	Feature	Fisher Score
1	Stemmed word "test"	0.94
2	Reported Category (`BUG`)	0.61
3	Reported Category (`TEST`)	0.50
4	Stemmed word "backport"	0.45
5	Stemmed word "remov"	0.38
6	Stemmed word "build"	0.37
7	Reported Category (`TASK`)	0.34
8	Stemmed word "eol"	0.33
9	Issue Reporter "brian.curnow@gfs.com"	0.33
10	Stemmed word "doap"	0.33

The y-axis corresponds to the ground truth labels. A cell in the confusion matrix indicates the number of issue reports of a particular ground truth category that is classified as a particular category by our approach. For example, from the confusion matrix, we can learn that 95 issues whose ground truth category is `DOCUMENTATION` are misclassified as `BUG`.

Table XI: Confusion Matrix: Ground Truth Versus Predicted Categories

	BUG	IMPR	BUILD	RFE	DOC	TASK	DESIGN-DEF	REFAC	SPEC	CLEANUP	TEST	BACKPORT	OTHERS
BUG	2631	119	8	48	23	0	14	1	4	8	26	1	31
IMPROVEMENT	320	658	13	214	12	2	6	19	0	16	8	0	4
BUILD SYSTEM	29	19	127	17	10	0	0	1	0	5	11	0	5
RFE	139	223	7	765	13	1	15	31	5	13	6	0	3
DOCUMENTATION	95	37	13	39	209	2	0	2	0	17	0	0	9
TASK	4	6	2	12	4	4	0	0	0	0	1	0	0
DESIGN DEFECT	74	18	0	36	0	1	33	11	0	4	1	0	0
REFACTORING	20	61	0	51	2	0	6	91	0	16	1	0	1
SPECIFICATION	54	6	0	8	0	0	0	0	13	1	1	0	1
CLEANUP	58	42	5	30	11	0	5	12	0	104	6	0	2
TEST	84	15	8	12	1	0	0	3	0	4	220	0	2
BACKPORT	12	3	0	3	1	0	0	0	0	2	1	4	0
OTHERS	84	17	6	9	9	0	0	2	0	0	1	0	25

Note: IMPR=IMPROVEMENT, BUILD=BUILD SYSTEM, DOC=DOCUMENTATION, DESIGN-DEF=DESIGN DEFECT, REFAC=REFACTORING, SPEC=SPECIFICATION

Table IX: Most Effective Features (Rhino)

No.	Feature	Fisher Score
1	Stemmed word "test"	3.84
2	Stemmed word "suit"	0.43
3	Stemmed word "patch"	0.32
4	Stemmed word "driver"	0.29
5	Stemmed word "regress"	0.27
6	Stemmed word "creat"	0.25
7	Stemmed word "rhino"	0.22
8	Reported Category (BUG)	0.20
9	Reported Category (RFE)	0.20
10	Issue Reporter "david"	0.20

Table X: Most Effective Features (Tomcat5)

No.	Feature	Fisher Score
1	Stemmed word "longer"	1.15
2	Issue Reporter "starksm"	0.71
3	Stemmed word "class"	0.64
4	Stemmed word "ant"	0.62
5	Reported Category (BUG)	0.56
6	Reported Category (RFE)	0.56
7	Stemmed word "extern"	0.56
8	Stemmed word "secur"	0.53
9	Stemmed word "outdis"	0.49
10	Issue Reporter "kroy"	0.49

From the confusion matrix, we can learn that issue reports of category BUG are often misclassified as IMPROVEMENT. Issue reports of category IMPROVEMENT are often misclassified as BUG and RFE. Issue reports of category RFE are often misclassified as IMPROVEMENT and BUG. Issue reports of category DOCUMENTATION are often misclassified as BUG. Issue reports of category REFACTORING are often misclassified as IMPROVEMENT and RFE. Issue reports of category CLEANUP are often misclassified as BUG and IMPROVEMENT. Issue reports of category TEST are often misclassified as BUG.

Moreover, issue reports of category BUILD SYSTEM is often well predicted. On the other hand, issue reports of categories TASK, DESIGN DEFECT, SPECIFICATION, and BACKPORT are often poorly predicted.

5) RQ5: Comparison to Other Classification Algorithms: We use Weka [16] to evaluate various other algorithms which support multi-class classification. We compare precision, recall and F-measure of these algorithms with our approach (SVM). We show the results in Table XII. We can note that SVM (i.e., LibSVM) outperforms the other multi-class classification algorithms.

D. Threats to Validity

Threats to internal validity relates to errors in our experiments. We have rechecked our code, still there could be errors that we missed. The validity of our ground truth depends on the reliability of the categories that are manually produced by Herzig et al. which is another threat to internal validity.

Threats to external validity relates to the generalizability of our findings. We have evaluated our approach using more than 7,000 issue reports from 5 systems. In the future, we plan to reduce this threat further by considering additional issue reports from more systems.

Threats to construct validity refers to the suitability of our evaluation metrics. We have used weighted precision, recall, and F-measure which are well known metrics and they have been used in many previous studies [17], [18]. Thus, we believe there is little threat to construct validity.

V. Related Work

In this section, we briefly describe related past studies. We first describe studies that also categorize bug reports. Next, we describe other studies that propose various ways to help developers manage bug reports. We then describe empirical studies on bug reports. Due to the space limitation, the survey here is by no means complete.

Table XII: Comparison with Other Algorithms

Approach	HTTPClient			Jackrabbit			Lucene-Java			Rhino			Tomcat5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
Ours (i.e., LibSVM)	0.61	0.63	0.60	0.71	0.72	0.71	0.62	0.63	0.62	0.58	0.61	0.57	0.58	0.62	0.58
Naive Bayes (NB)	0.49	0.47	0.48	0.51	0.39	0.43	0.46	0.37	0.40	0.51	0.51	0.51	0.48	0.40	0.42
NB Multinomial	0.53	0.60	0.54	0.64	0.66	0.61	0.60	0.59	0.56	0.52	0.58	0.49	0.51	0.58	0.47
K-Nearest Neighbors	0.47	0.29	0.34	0.60	0.58	0.59	0.46	0.40	0.42	0.50	0.43	0.43	0.43	0.43	0.42
Random Forest	0.45	0.56	0.46	0.54	0.58	0.53	0.45	0.48	0.43	0.51	0.56	0.47	0.45	0.56	0.46
RBF Network	0.37	0.39	0.37	0.39	0.41	0.40	0.31	0.31	0.30	0.40	0.43	0.41	0.33	0.54	0.39

A. Categorization of Issue/Bug Reports

The closest work to ours is the work by Antoniol et al. which predicts if an issue report is a bug report or an enhancement [4]. In this work, we extend Antoniol et al.'s work by considering *fine granularity* category prediction, i.e., we predict not one out of two categories, but *one out of thirteen categories*, which is a harder problem. We also consider the *reclassification setting* where our goal is to reclassify the categories of issue reports by taking into consideration the original reported categories of the reports along with other features.

There are also other studies that also employ classification algorithms to categorize bugs into various labels. Gegick et al. use a text classification approach to predict if a bug report is a security bug or not [19]. Their model was able to classify 78% of the security bug reports which were labeled as not-security bug reports. Thung et al. propose a method to automatically categorize bug reports into either a control and data flow bug, or a structural bug [20]. Their results show that automatic classification using multiclass classification algorithm can label defects with an accuracy of 77.8%. Menzies and Marcus present an approach that predict fine-grained severity labels of bug reports [21]. Their work is extended by Lamkanfi et al. which categorize bug reports into two categories: severe and not severe [22]. They evaluate their approach on bug reports from a number of open source projects. As a follow up to their previous work, Lamkanfi et al. also investigate the performance of a number of classification algorithms to predict bug reports severity and show that Naive Bayes Multinomial performs the best [23]. Tian et al. predict fine-grained severity labels by using an extended BM25 to measure similarity between bug reports and a nearest neighbor classification algorithm [24]. Huang et al. classify the impact of defects by analyzing textual features extracted from bug reports [25]. Using these features, they assign bug reports into one of the following category labels: reliability, capability, integrity, usability, and requirements category labels. They show that they can achieve F-measure scores of 0.222, 0.885, 0.700, 0.629, and 0.393, for each of the 5 category labels, respectively. Hindle et al. propose an automated approach to assign large changes into several maintenance categories [26]. They have experimented with various classification algorithms and they show that these algorithms can achieve accuracy scores of 13-70%.

B. Bug Report Management

There are other studies that propose ways to help developers deal with a large number of bug reports. We highlight past studies on duplicate bug report detection and bug triaging.

Bug reporting is an uncoordinated and distributed process. Thus, often multiple reports are made which describe the same problem. These bug reports are referred to as duplicate bug reports. A number of studies have been proposed to detect duplicate bug reports. Most of these studies take a bug report and recommend top-k most similar older reports to it. Runeson et al. extract textual contents from bug reports and use various measures (e.g., cosine, dice and jaccard) to compute the similarity of two bug reports [27]. The results of their study shows that 2/3 of the duplicate bug reports can be found using natural language processing techniques. Wang *et al.* extend Runeson et al.'s work by augmenting the textual information with execution trace information [28]. They show that the execution trace information, when available, can improve performance. Sun *et al.* propose a classification-based approach and extend BM25F to retrieve duplicate reports [29], [30]. Our work is orthogonal to the above as we aim to classify the type of an issue report.

Bug triaging is the task of assigning a bug report to the right developer to fix it [31], [32], [33], [34], [35]. Anvik et al. and Cubranic et al. propose the usage of machine learning algorithms such as Naive Bayes, SVM, and C4.5 to solve bug triaging problem [31], [32]. More recently, Tamrawi et al. propose an algorithm named Bugzie which uses fuzzy set theory to recommend a bug report to an appropriate fixer [33]. The proposed approach keep in its cache the terms that characterize each developer and uses this cache to measure the suitability of a developer to a new bug report. Jeong et al. investigate the reassignments of bug report fixers (aka. bug report tossing) in Mozilla and Eclipse, and propose a graph-based approach that uses the concept of Markov chain to improve bug triaging [34]. Their model can help to find team structures, find experts and assign developers to bug reports. Their technique can reduce the tossing event by 72% and improve prediction accuracy by upto 23%. Bhattacharya et al. use several techniques such as refined classification using additional attributes, ranking function for the potential tossees and multi-feature tossing graphs [35]. Their technique can achieve 83.62% prediction accuracy in bug triaging and reduce the tossing path lengths to 1.5-2 tosses.

C. Empirical Studies of Bug Reports

Researchers have also done empirical studies on bug repositories (aka. issue tracking system). Anvik et al. perform a study that investigates the characteristics of a number of bug repositories [1]. In their paper, they show the number of reports submitted by developers and the percentages of different bug resolutions. They also show that developers are overwhelmed with the large number of bug reports in repositories. Sandusky et al. analyze bug report networks in open source development communities and report the nature, impact, and extent of these networks [36]. Hooimeijer and Weimer predict bug report quality by building a statistical model based on various features that are extracted from more than 27,000 bug reports in open source projects [37]. Their model predicts whether a bug report is triaged within the stipulated time and can reduce the cost of software maintenance in specific situations. Bettenburg et al. reports what makes a good bug report by surveying developers of Eclipse, Mozilla, and Apache [38]. Their study finds that a good bug report is one that provides enough information for developers to perform debugging activities. Most recently, Herzig et al. investigate more than 7,000 issue reports and manually reclassify the categories of these issue reports [3]. They highlight that misclassification impacts the task of predicting bug-prone files.

VI. Conclusion and Future Work

In this work, we propose an approach that predicts if an issue report is misclassified, and if so, outputs its new predicted category. Our classification based approach extracts a number of feature values from a training set of issue reports, and creates a statistical model based on these feature values and ground truth categories. The extracted features include pre-processed words that appear in the summary and description fields, presence of stack trace, issue reporter identifier, and the reported issue category. Based on the values of these features, we use LibSVM, which is a multi-label classification algorithm, to create a statistical model. This model is then used to predict the category that should be assigned to a new issue report. We have evaluated our approach on more than 7,000 issue reports from 5 software systems and the results show that our approach can achieve weighted F-measure scores of 0.57-0.71 which improves the results of several baseline approaches by 28.88%-414.66%.

As future work, we plan to design more advanced multi-class classification solution that can achieve higher F-measure scores. We also plan to reduce the threats to external validity further by investigating more issue reports from more systems.

Acknowledgement

We would like to thank Kim Herzig, Sascha Just, and Andreas Zeller for making their issue report datasets publicly available.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, pp. 35–39, 2005.

[2] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *CSMR-WCRE*, pp. 174–183, 2014.

[3] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *ICSE*, pp. 392–401, 2013.

[4] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, pp. 23:304–23:318, 2008.

[5] M. F. Porter, "Readings in information retrieval," ch. An Algorithm for Suffix Stripping, Morgan Kaufmann Publishers Inc., 1997.

[6] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, *et al.*, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2008.

[7] A. McCallum, K. Nigam, *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752, pp. 41–48, Citeseer, 1998.

[8] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, pp. 249–258, 2011.

[9] A. Prinzie and D. V. den Poel, "Random forests for multiclass classification: Random multinomial logit," *Expert Systems with Applications*, vol. 34, no. 3, 2008.

[10] D. Broomhead and D. Lowe, "Radial basis functions, multi-variable functional interpolation and adaptive networks," tech. rep., DTIC Document, 1988.

[11] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[12] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[13] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *KDD*, pp. 557–566, 2009.

[14] J. Han and M. Kamber, *Data Mining Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2006.

[15] R. Duda, P. Hart, and D. Stork., *Pattern Classification*. Wiley Interscience, 2000.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *ACM SIGKDD Explorations Newsletter*, vol. 11, pp. 10–18, 2009.

[17] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[18] R. Fernández, J. Ginzburg, and S. Lappin, "Classifying non-sentential utterances in dialogue: A machine learning approach," *Computational Linguistics*, vol. 33, no. 3, pp. 397–427, 2007.

[19] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *MSR*, pp. 11–20, 2010.

[20] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCRE*, pp. 205–214, 2012.

[21] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, pp. 346–355, 2008.

[22] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, pp. 1–10, 2010.

[23] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, pp. 249–258, 2011.

[24] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *WCRE*, pp. 215–224, 2012.

[25] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *ASE*, pp. 412–415, 2011.

[26] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *ICPC*, pp. 30–39, 2009.

[27] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, pp. 499–510, 2007.

[28] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, pp. 461–470, 2008.

[29] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, pp. 45–54, 2010.

[30] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, pp. 253–262, 2011.

[31] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *ICSE*, pp. 361–370, 2006.

[32] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *SEKE*, pp. 92–97, 2004.

[33] A. Tamrawi, T. Nguyen, J. Al-Kofahi, and T. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *ESEC/FSE*, pp. 365–375, 2011.

[34] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *ESEC/FSE*, pp. 111–120, 2009.

[35] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *ICSM*, pp. 1–10, 2010.

[36] R. J. Sandusky, L. Gasser, and G. Ripoche, "Bug report networks: Varieties, strategies, and impacts in a f/oss development community," in *MSR*, pp. 80–84, 2004.

[37] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *ASE*, pp. 34–43, 2007.

[38] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *ESEC/FSE*, pp. 308–318, 2008.