Innovations in Systems and Software Engineering⋆ manuscript No.
(will be inserted by the editor)

Automated Classification of Software Issue Reports Using Machine Learning Techniques: An Empirical Study

Nitish Pandey · Debarshi Kumar Sanyal · Abir Hudait · Amitava Sen

Received: 19-03-2017 / Accepted: 28-06-2017

Abstract Software developers, testers and customers routinely submit issue reports to software issue trackers to record the problems they face in using a software. The issues are then directed to appropriate experts for analysis and fixing. However, submitters often misclassify an improvement request as a bug and vice versa. This costs valuable developer time. Hence automated classification of the submitted reports would be of great practical utility. In this paper we analyze how machine learning techniques may be used to perform this task. We apply different classification algorithms, namely naive Bayes (NB), linear discriminant analysis (LDA), k-nearest neighbors (kNN), support vector machine (SVM) with various kernels, decision tree (DT) and random forest (RF) separately to classify the reports from three open-source projects. We evaluate their performance in terms of F-measure, average accuracy and weighted average F-measure. Our experiments show that random forests perform best while SVM with certain kernels also achieve high performance.

⋆ This is a postprint of the paper with DOI 10.1007/s11334-017-0294-1 in Innovations Syst Softw Eng (2017) 13:279-297

Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait
School of Computer Engineering, KIIT University, Bhubaneswar–751024, Odisha, INDIA
E-mail: {nitish5808, debarshisanyal, abirhudait}@gmail.com (in order)
Debarshi Kumar Sanyal is currently on lien from KIIT University and working as Research Consultant at Indian Institute of Technology Kharagpur, Kharagpur–721302, West Bengal, INDIA. He is the corresponding author.

Amitava Sen
Department of Computer Science and Engineering, JIS University, Kolkata–700109, West Bengal, INDIA
E-mail: amitavasen@yahoo.com

Keywords Bug classification · Machine learning · F-measure · Accuracy · Random forest

1 Introduction

A software evolves continuously over its lifetime. As it is developed and maintained, bugs are filed, assigned to developers and fixed. Bugs can be filed by developers themselves, testers or customers, or, in other words by any user of the software. For open-source projects, defect tracking tools like Bugzilla[1], Gnats[2] and Jira[3] are commonly used for storing bug reports and tracking them till closure. Proprietary software also uses same or similar tools. However, along with bugs (meant for corrective maintenance of the software), it is common to file change requests that ask for adaptation of the software to new platforms (adaptive maintenance) or to incorporate new features (perfective maintenance). Add to that requests to update the documentation or refactor the code, or simply discussions and requests for help. Given this diversity, the person filing the issue may not always make a fine-grained distinction between the different kinds of reports and instead file them as bugs only. In fact, research shows misclassifications are commonplace [2, 14]. In an elaborate study involving more than 7000 issues spanning 5 projects, researchers found that 33.8% of all reports are misclassified [14]. The consequence could be costly: developers must spend their precious time to look into the reports and relabel them correctly. Hence it is worthwhile to explore if this classification can be done automatically.

[1] https://www.bugzilla.org/
[2] https://www.gnu.org/software/gnats/
[3] https://www.atlassian.com/software/jira/

We call a report in a software defect tracking system an *issue report* irrespective of whether it refers to a valid bug or it is some other concern (as discussed above). In this paper, we study how supervised learning techniques [30] can be used to automatically classify an issue report as referring to a bug or to something that is *not* a bug. The key intuition is, certain terms that describe errors or failures in the software are more common in descriptions that truly report a bug. Thus a classifier could be trained with a set of issue reports, each of which contains some description of the issue and is already labeled with the correct category. It can then proceed to classify an unlabeled report that it has not seen at training time.

A preliminary version of this work appears in a conference paper [27]. The main extensions include addition of new classifiers, larger datasets, more experiments and detailed discussion on various aspects.

Contribution: In this paper, we report our experiences with application of machine learning algorithms to classify issue reports from JIRA into bug and non-bug categories in a completely automated way. The issue reports selected belong to three open-source projects HTTPCLIENT[4], LUCENE[5] and JACKRABBIT[6]. We use the following classification algorithms as implemented in R[7]: naive Bayes (NB), linear discriminant analysis (LDA), k-nearest neighbors (kNN), support vector machine (SVM) with various kernels, decision tree (DT) and random forest (RF) separately. We analyze the effect of different parameters on the classifiers' predictive power. We measure classification efficiency in terms of F-measure, average accuracy and weighted average F-measure. Highest F-measure varies within $69-76\%$ while both highest average classification accuracy and highest weighted average F-measure values vary within $75-83\%$ depending on the project. This boosts our belief that machine learning tools can be effectively used as a first level filter on issue reports before they are escalated to the developers.We also show experimentally the critical need for manually classified issue reports in classifier training to achieve high quality automated classification.

Roadmap: The paper is organized as follows. The background of our study and the formal problem definition are given in section 2. Preliminaries on supervised machine learning appear in section 3. Our proposed approach is outlined in section 4 followed by implementation details in section 5. Experiments and results along with the threats to their validity are discussed in section 6. Related work is reported in section 7. The paper concludes in section 8.

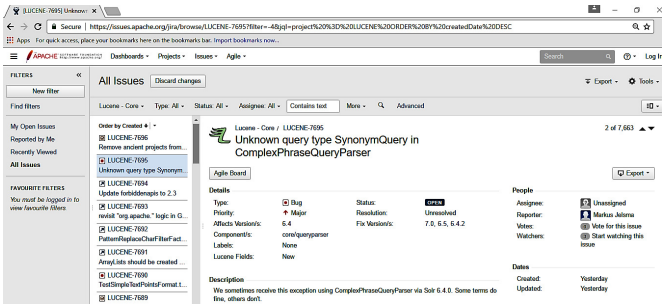

Fig. 1 LUCENE issue reports in JIRA

2 Background and Problem Statement

2.1 Background

Issue reports capture crucial information about the problems faced by the users of a software. A variety of issue tracking tools are available, with varying degrees of sophistication in handling the issue filed. The variations occur in the number of fields the user needs to fill in to the number of stages the issue goes through before it is declared closed. In this pilot study, we consider issue reports from JIRA. It is a proprietary issue tracking system developed by Atlassian in 2002. Atlassion provides JIRA for free to open-source projects that meet certain criteria. It is used by Fedora Commons, Hibernate, Skype and the Apache Software Foundation among others. An issue in JIRA may report a host of different things. Some of the common categories (called `Type` in JIRA) include `BUG` (that call for corrective maintenance), `IMPR` (denoting requests for improvement of the existing features), `RFE` (that asks for new features), `DOC` (requiring updates to documents like user manuals), `REFAC` (suggesting code refactoring) and `OTHER` (includes back-porting software, adding test cases, etc.). Although such a granular differentiation is useful for issue triaging and tracking, in this paper we will group issues into only two types, `BUG` and `NUG` where `NUG` includes all categories except `BUG`, i.e., problems that do not require corrective maintenance. This simplification makes our study easier and, we believe, is acceptable as a first level sieve for the reports. An issue report (see Figure 1) also contains other fields like `Summary` (usually a one-line gist of the problem), `Description` (detailed account of the problem including stack traces and test cases), `Priority`, `Affects Version/s`, `Component/s`, etc.

A submitter may not always have a thorough knowledge of the specification of the software or the environ-

[4] `hc.apache.org/httpclient-3.x/`
[5] `https://lucene.apache.org/`
[6] `http://jackrabbit.apache.org/jcr/index.html`
[7] `https://www.r-project.org/about.html`

ment where it is running. So she often leaves some fields of the report empty [14]. More unfortunately, she may not be able to make a quick and correct distinction between bugs and non-bugs. A genuine bug should receive higher priority than other requests; therefore a misclassification may lead to incorrect focus on the issue from the developers. Manual classification by application engineers or developers after the issue has been filed requires a thorough review of the problem and consumes considerable time [14]. We feel it is better if automated classification can be done to categorize the issues into `BUG` and `NUG`. Such a classifier could be integrated with the tracker and monitor every issue being filed. If it thinks the issue is wrongly classified by the submitter, it may prompt her with a suggestion. Our belief is strengthened from the success of machine learning tools in various applications like spam detection, text classification and image recognition. Here a classifier is first trained with a set of labeled instances and then asked to recognize the class of a new instance.

We choose a number of supervised learning algorithms and explore if they can be used to construct classifiers that can automatically identify if an issue is a `BUG` or `NUG`. For our experiments, we use a subset of the corpus[8] curated by [14]. The corpus contains *manually* classified 7401 issue reports from five open-source projects; three of these projects are tracked through JIRA and two through BUGZILLA. These manually classified reports are immensely useful for automated classification since misclassified instances lead to bias in training and consequent incorrect outputs on test instances. The corpus is organized as a set of 5 csv files, each corresponding to a single project. Every row in a csv file is a 3-tuple: `(ID, CLASSIFIED, TYPE)` referring respectively to the report identifier (that links to its full information in the repository), the report type that the researchers considered most appropriate and the type as mentioned in the repository. In many cases, `CLASSIFIED` and `TYPE` do not agree, pointing to a misclassification in the repository. For our study, we choose only the reports extracted from JIRA. They belong to three prominent open-source projects from Apache, namely, HTTPCLIENT (that aims to extend and provide robust support to the base HTTP protocol; it is no longer being developed and has been replaced by the Apache HTTPCOMPONENTS[9] project), LUCENE (that is a high-performance, full-featured text search engine library coded in JAVA) and JACKRABBIT (which is an open source content repository for the JAVA platform). We use JIRA and the corpus to reconstruct the issue details. However, our representation is concise; we represent each report with only 4 fields: (`ID`, `CLASSIFIED`, `TYPE`, `SUMMARY`) where the first 3 fields are the same as in the above corpus (with values in `CLASSIFIED` and `TYPE` mapped to {`BUG`, `NUG`}; some issues, though, have missing values in `TYPE` field) and the fourth same as `Summary` in JIRA. Other fields like `Priority`, `Description`, `Reporter`, etc. present in JIRA are ignored. Previous researchers have shown that an issue summary has almost the same information content as longer descriptions but with less noise and can be relatively accurately parsed [19, 17, 37]. So we consider the summary part and ignore the longer description for automated analysis.

Fig. 2 Commonality cloud showing 100 most frequently occurring terms common to the summaries of `BUG` and `NUG` categories in HTTPCLIENT.

A cursory look at the summaries of the collected reports from HTTPCLIENT reveals two distinct facets. First, as Figure 2 shows, the summaries in each category share many common terms. Second, and fortunately, as Figure 3 shows, the most frequent terms in a `BUG` report are not always the most frequent ones in a `NUG` report. The second observation suggests issue reports may be classified using text classification-like techniques applied on issue summaries. Figure 4 shows that within a large enough set of the most frequent terms in HTTPCLIENT summaries, the top discriminative terms from both `BUG` and `NUG` categories are present. So it might suffice to choose a set of most frequent terms without regard to the issue categories. Intuitively, given an issue whose actual type we seek to determine, we should examine which of these terms appear in its summary. Unlike some previous researchers who applied topic modeling on issue reports (summary and description combined) [28], we simply represent each summary as a bag-of-words and classify it based on the frequency of different terms in it. Recently, Zhou *et al.* [40] have used fields other than summary and description and advocated a two-phase classification scheme. But we do not include these fields since one of our objectives is to keep the approach very simple.

[8] available at `https://www.st.cs.uni-saarland.de/softevo//bugclassify/`

[9] `http://hc.apache.org/`

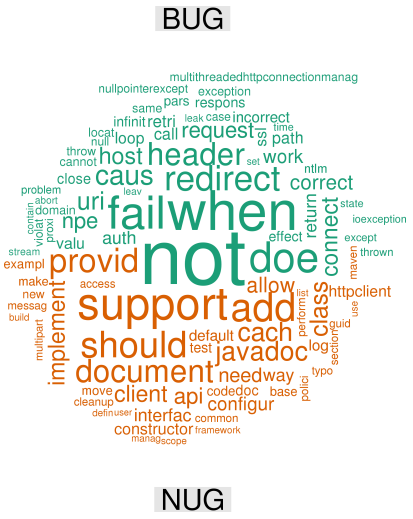

Fig. 3 Comparison cloud showing top 100 discriminant terms in the summaries of `BUG` and `NUG` categories of HttpClient respectively.

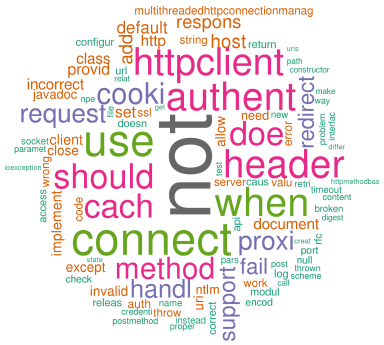

Fig. 4 Word cloud showing 100 most frequent terms in the summaries of HttpClient issues. Here we ignore the category (`BUG` / `NUG`) of a summary.

2.2 Problem Statement

We are given a set $\wp$ of classifiers, each of which uses supervised learning to classify instances. The instances, in our case, comprise software issue reports extracted from an issue tracker. Each report belongs to one of the two classes in $\mathcal{C}$ where $\mathcal{C} = \{c_1, c_2\}$; c_1 represents `BUG` and c_2 represents `NUG`. The set of instances is denoted by $\mathcal{X}$. This set is used to train and test the classifiers in $\wp$. Our aim is to estimate the predictive power of each classifier for given classifier configurations. We also intend to find out how various parameters like feature vector size influence classifier performance.

3 Preliminaries

We sketch a brief overview of supervised learning techniques and various classifiers used in the paper.

3.1 Supervised Learning Techniques

We are presented with a set $\mathcal{X}$ of instances. Each instance $\mathbf{x}_i$ is represented by a p-dimensional vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})(p \geq 1)$ where $x_{ij}(1 \leq j \leq p)$ is a single feature. Each instance belongs to exactly one of the classes in the set $\mathcal{C}$. A *classifier* is a function $f : \mathcal{X} \rightarrow \mathcal{C}$. A training set is defined as a set of m labeled instances $\mathcal{T} = \{(\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \ldots, (\mathbf{x}_m, c_m)\}$ where $c_i(1 \leq i \leq m) \in \mathcal{C}$ is the class label of the ith instance $\mathbf{x}_i$. A *supervised learning algorithm* is an algorithm that outputs a classifier f after reading the training set. The classifier is used to determine the label of an instance that has not been encountered during training. Note that supervised learning can be used for classification as well as regression; in case of classification, the labels are categorical data. We are interested in binary classification, i.e., $|\mathcal{C}| = 2$. Several state-of-the-art classification algorithms are available in data mining literature. We briefly discuss the principles behind the ones we use in this paper.

Naive Bayes (NB): NB classifier uses Bayes' theorem to classify problem instances after it has been trained. It assumes that the values of the features are independent of each other, given a class label. Thus even though the values of the different features may have some correlation, NB ignores it. Suppose, as above, a problem instance is represented by a feature vector $\mathbf{x}$ such that its constituent features are independent within its true class. Also suppose the class of an instance is denoted by c_k where $k \in \{1, 2\}$. Bayes' theorem computes the conditional probability of the instance belonging to class c_k as, $P(C = c_k|\mathbf{X})$ [10] $= \frac{P(C=c_k)P(\mathbf{X}|C=c_k)}{P(\mathbf{X})}$ where $C, \mathbf{X}$ are random variables denoting the class and the feature vector of the instance respectively. It labels the instance with the class c_k for which the posterior $P(C = c_k|\mathbf{X})$ is maximized. In practice, only the numerator is important so that the posterior is simply the prior (i.e., $P(C = c_k)$) times the likelihood (i.e., $P(\mathbf{X}|C = c_k)$). The fraction of different class labels in the training subset can be used to estimate $P(C)$. For continuous feature values, Gaussian NB assumes the values are distributed within class according to a normal distribution, i.e., $X_i|C \sim \mathcal{N}(\mu_{iC}, \sigma^2_{iC})\forall i \in \{1, 2, \ldots, n\}$[11]. The training data is used to learn the parameters of this distribution. Since the features are assumed independent within the true class, $P(\mathbf{X}|C = c_k) = \prod_{1 \leq i \leq p} P(X_i|C = c_k)$. For discrete feature values, multinomial NB is generally appropriate. The main

[10] denotes $P(C = c_k|\mathbf{X} = \mathbf{x})$

[11] $\mathcal{N}(\mu, \sigma^2)$ denotes normal distribution with mean μ and variance σ^2

pros of NB are that it is fast, has low space requirements and is robust against irrelevant features.

Linear Discriminant Analysis (LDA): LDA also uses Bayes' theorem for classification. But instead of assuming that the feature vectors are conditionally independent given the class, LDA models the feature vectors in a class as a Gaussian distribution with a mean vector and a covariance matrix. Further, it assumes that the covariance matrices are identical for all classes and of full rank, i.e., $\mathbf{X}|C \sim \mathcal{N}(\boldsymbol{\mu}_C, \boldsymbol{\Sigma})$. This makes the decision boundary between the classes linear in the input feature vector $\mathbf{x}$. It labels a new instance with the class c_k whose mean vector is *closest* to it in the sense of Mahalanobis distance.

k-Nearest Neighbors (kNN): A kNN classifier follows a non-parametric method; during training, it simply stores the training subset without learning any explicit function from it. When faced with a new instance to classify, it simply assigns to it the class label that is most frequent among the k training instances *closest* (e.g., in terms of Euclidean distance) to it. Here k is a user input. Large k may lead to underfitting while small k may result in overfitting. So proper choice of k is essential. Though simple, kNN has a costly prediction time and storage requirement.

Support Vector Machine (SVM): A linear SVM is a non-probabilistic classifier that, given a training subset with p-dimensional feature vectors belonging to two classes c_1 and c_2, attempts to find a maximum margin $(p-1)$-dimensional hyperplane that separates the classes. In other words, the hyperplane has the maximum distance from the nearest training instances on its either side. However, if the sets are not linearly separable in p-dimensional space, linear SVM will not be able to separate them perfectly. A standard technique is to map the feature vector to a higher dimensional space via a *kernel* function and construct a hyperplane in that space. Common kernels include radial basis function, polynomial and sigmoid function. SVMs are generally slower compared to the classifiers discussed above but are known for their accuracy.

Decision Tree(DT): A DT is a classifier which depicts in a tree form the process of decision making during classification. The class labels are represented by the leaf nodes. An internal node specifies some test to be carried out on a single feature (of the feature vector). A branch out of an internal node identifies an outcome of the test and may lead either to a leaf node or to another internal node. Given a test instance $\mathbf{x}$, one starts at the root of the tree and moves through it until a leaf node is reached, whence the output class label is determined.

Random Forest (RF): RF is an ensemble learning method for classification [6]. It grows a number of decision trees at training time. Each decision tree is built with a different bootstrap sample drawn from the training subset and is constructed to the maximum size without any pruning. A randomly selected subset of features is used to split each node in a tree. A test instance $\mathbf{x}$ to be classified is run through each decision tree and is assigned the class that is the mode of the classes reported by the individual trees in the forest. RFs do not overfit data, are fast and highly accurate on large datasets [6].

3.2 Performance Measures

To measure the performance of the classifiers, we use the measures of precision, recall, F-measure and accuracy. Before we define them, we look at four important quantities defined with respect to a class of interest.

1. *True Positive (TP):* Number of reports correctly labeled to a class.
2. *True Negative (TN):* Number of reports correctly rejected from a class.
3. *False Positive (FP):* Number of reports incorrectly labeled to a class.
4. *False Negative (FN):* Number of reports incorrectly rejected from a class.

The entries of the confusion matrix, in terms of the above vocabulary, are indicated below in Table 1. We refer to `BUG` as the class of interest. The complementary class is `NUG`. While we definitely want high accuracy, i.e., `BUG`s classified as `BUG`s and `NUG`s classified as `NUG`s, we are also interested in optimizing other measures that reduce false positives and false negatives with respect to identification of `BUG`s. The various measures we are interested in are described next.

Table 1 Confusion matrix of a classifier

		Results of classifier	
		`BUG`	`NUG`
True classification	`BUG`	TP	FN
	`NUG`	FP	TN

Precision: It is the ratio of the number of true positives to the total number of reports labeled by the classifier as belonging to the positive class.

$$Pr = \frac{TP}{TP + FP} \tag{1}$$

Recall: It is the ratio of the number of true positives to the total number of reports that actually belong to the positive class.

$$Re = \frac{TP}{TP + FN} \tag{2}$$

F-Measure or F_1-score: It is the harmonic mean of precision and recall.

$$F = 2 * \frac{Pe * Re}{Pe + Re} = \frac{2TP}{2TP + FP + FN} \tag{3}$$

Accuracy: It measures how correctly the classifier labeled the records.

$$Acc = \frac{TP + TN}{TP + FP + FN + TN} \tag{4}$$

Measures for cross-validation: In k-fold cross-validation, we divide the available training set $\mathcal{T}$ into k disjoint subsets $\mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots, \mathcal{T}^{(k)}$ of equal size. Each subset $\mathcal{T}^{(i)} (1 \leq i \leq k)$ is used as a test subset and its performance measured against a classifier $f^{(i)}$ trained with $\mathcal{T} \setminus \mathcal{T}^{(i)}$. Researchers are usually interested to estimate the performance of the classifier in terms of F-measure and accuracy when the learning algorithm is run on the entire set $\mathcal{T}$. But there is some divergence of views in the literature on how to aggregate the measurements of a quantity in different folds into a single *average* value. We resolve in favor of the recommendation in [11]. Assuming $B^{(i)}$ denotes the value of a quantity calculated in fold i, we define B as its aggregate over the k folds. In particular, we define the following.

$$\begin{aligned} TP &= \sum_{i=1}^{k} TP^{(i)} \\ FP &= \sum_{i=1}^{k} FP^{(i)} \\ TN &= \sum_{i=1}^{k} TN^{(i)} \\ FN &= \sum_{i=1}^{k} FN^{(i)} \end{aligned} \tag{5}$$

Precision, recall and F-measure over the k folds are now defined using expressions 1, 2 and 3 respectively. If there are only 2 classes c_1 and c_2 with n_1 and n_2 instances and F-measures F_{c_1} and F_{c_2} respectively, the weighted average F-measure [40] is given by

$$\widehat{F} = \frac{n_1 F_{c_1} + n_2 F_{c_2}}{n_1 + n_2} \tag{6}$$

The average accuracy over k folds is the mean of the accuracies in each fold [11].

$$\overline{Acc} = \frac{1}{k} \sum_{i=1}^{k} Acc^{(i)} \tag{7}$$

4 Proposed Approach

We classify issue reports from Jira into two classes, `BUG` and `NUG`. Our approach is shown schematically in Figure 5. As already mentioned, we collect issue reports that have been manually assigned appropriate category. Each report is a 4-tuple: `(ID, CLASSIFIED, TYPE, SUMMARY)` where `ID` is the unique identifier for the issue in Jira, `CLASSIFIED` is the type assigned by manual review [14], `TYPE` is the issue type in Jira and `SUMMARY` is a short (one-line) textual description of the issue in Jira.

Generation of training and test subsets: We use k-fold cross-validation to measure the predictive power of a classifier. In order to generate k folds, k iterations are run. In each iteration, stratified sampling is used to generate a test subset. If there are N reports in a project, each of the first $k-1$ test subsets contain $\lfloor N/k \rfloor$ instances while the k-th test subset holds $N - (k-1) * \lfloor N/k \rfloor$ instances. The test subsets do not overlap. Thus each instance is tested exactly once. In an iteration, all instances not in the test subset form the training subset.

Text preprocessing and DTM generation: The training process starts with preprocessing the collection of issue reports in the training subset. In preprocessing, text is converted to lower case followed by removal of punctuations, numbers and stopwords. Note that our list of stopwords is very limited and comprises mainly articles, conjunctions, prepositions and pronouns. In particular, we do not remove stopwords denoting negatives like "not" and modal auxiliaries like "should" that influence the meaning of the reports significantly. For example, a bug report is more likely to have negatives like "not" while an improvement request can specify a desirable feature with "should". Similarly temporal connectives like "before", "after", "when", etc. usually describe test scenarios used to report issues. Hence they are also not eliminated. The resultant corpus is then stemmed and converted to a document term matrix (DTM). The rows of a DTM identify documents while the columns represent terms that occur in the documents. An entry d_{ij} in a DTM indicates the frequency of term j in document i. In our case, a document corresponds to the summary of a single issue.

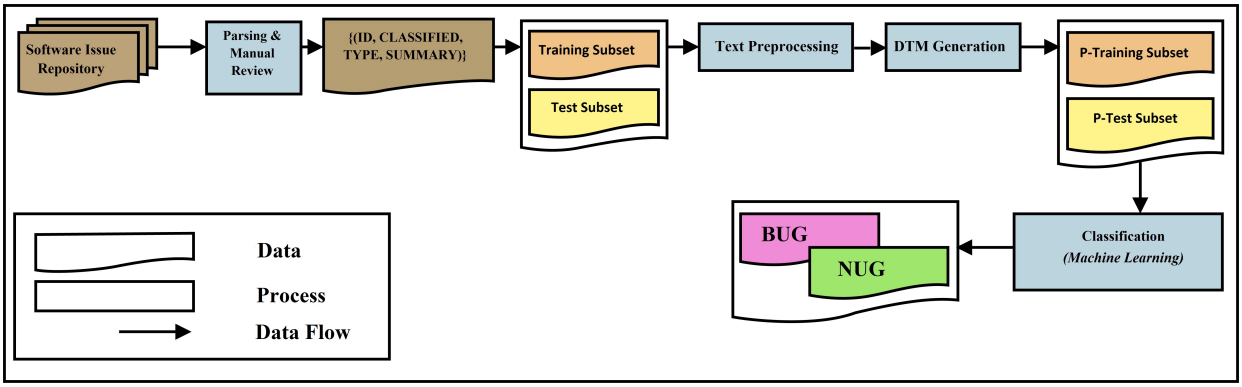

Fig. 5 Proposed approach for automated classification of issue reports. k-fold cross-validation is used.

```
#Examples of two issue reports
ID        CLASSIFIED  TYPE   SUMMARY
JCR-450   BUG         BUG    Memory leak in UUIDDocId
JCR-212   NUG         NUG    decorator enhancements

#DTM for the above summaries
Docs  Terms
----  -----------------------------------
      decor  enhanc  leak  memori  uuiddocid
1     0      0       1     1       1
2     1      1       0     0       0

#DTM with CLASSIFIED value prepended
Docs CLASSIFIED  Terms
---- ----------  -----------------------------------
                 decor  enhanc  leak  memori  uuiddocid
1    BUG         0      0       1     1       1
2    NUG         1      1       0     0       0
```

Next we normalize the raw term frequencies to model the term distribution better. We define the following term weights [29].

$W1$. d_{ij} is kept unchanged, i.e., raw term frequencies are used.

$W2$.

$$d_{ij} = \begin{cases} 1 + \log d_{ij}, & \text{if } d_{ij} \neq 0 \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

It handles burstiness in term distribution in a document while being an identity transform for zero and unit frequencies.

$W3$.

$$d_{ij} = \alpha + (1-\alpha)\frac{d_{ij}}{\max_k d_{ik}}; 0 \leq \alpha < 1 \tag{9}$$

We set default value of $\alpha = 0$. This transform simply normalizes each term frequency with the maximum term frequency in the document and thus prevents bias towards longer documents. Every score d_{ij} now lies in $[\alpha, 1]$.

We allow the term weights to be further normalized by document length measured using standard vector space model of document representation. We define the following normalizers based on document length [29].

$L1$. d_{ij} is kept unchanged.

$L2$.

$$d_{ij} = \frac{d_{ij}}{\sqrt{\Sigma_k d_{ik}^2}} \tag{10}$$

It normalizes frequencies with the document length in vector space model, thus preventing any single document from influencing the learning significantly. In fact the cosine term $\sqrt{\Sigma_k d_{ik}^2}$ roughly measures the number of unique terms in document i.

We now define the following transforms by combination of the above transforms (here, $Z = X + Y$ means transformation Z is formed by first applying transformation X on the term weights $\{d_{ij}\}$ and then applying Y on the above transformed term weights). (1) $T1 = W1 + L1$; this corresponds to no transform, i.e., raw term frequencies are used, (2) $T2 = W1 + L2$, (3) $T3 = W2 + L1$ (4) $T4 = W2 + L2$, (5) $T5 = W3 + L1$, (6) $T6 = W3 + L2$. We also define another transform $T7$ as follows.

$$d_{ij} = \frac{d_{ij}}{\Sigma_k d_{ik}} \tag{11}$$

where d_{ij} is the raw term frequency. It normalizes frequencies with the total frequency of all terms in the document.

We do not weigh terms in the DTM with term frequency-inverse document frequency (*tf-idf*) since certain terms are really expected to occur more frequently in `BUG` class while others more commonly in `NUG` class. Similar opinion is also expressed in [2].

Usually the number of terms in a DTM is far larger than the number of documents in the corpus since the terms in the issue summaries do not adhere to a controlled vocabulary. For example, in the LUCENE dataset,

a typical training subset generated for 10-fold cross-validation has 2197 reports with a total of 2973 terms. A DTM is generally very sparse, i.e., most of the columns in a row contain zeroes. To remedy this, we prune the DTM by removing terms that are very infrequent in the DTM. The number of terms to be retained in the DTM is a configurable parameter in our implementation. Reducing the dimensions also speeds up classifier training to be done later. Next, we build a dictionary of terms present in the training DTM. Subsequently we construct a DTM from the test subset, but include only those terms that are present in the dictionary because newer terms would not be identifiable to the classifier. For normalizing the entries of test DTM, we need values of $\max_k d_{ik}$, $\sqrt{\Sigma_k d_{ik}^2}$ and $\Sigma_k d_{ik}$. Here we make k iterate over all terms in document i and not just over the terms that are present in the dictionary. Finally, the correct issue type identified by the value of `CLASSIFIED` is added to each row of each DTM to compose the preprocessed training and test subsets or simply, *p-training and p-test subsets* respectively. The class label in p-training subset is needed for supervised learning in the training phase. For p-test subset, it is used later to check how accurately the classifier could label the instances.

Machine learning: This phase consists of training each classifier using the p-training subset and testing it on the p-test subset. This is done for the k pairs of training and test subsets generated by stratified sampling. Finally, the precision, recall, F-measure, weighted average F-measure and average accuracy over the k folds are reported for each classifier using expressions 1, 2, 3, 6, 7 respectively.

5 Implementation

We have developed an R script `BugBang.R` to predict the performance of the classifiers. We have used R version 3.2.1.

Input and output: Apart from the input and the output filenames (with absolute path), the R script accepts values of a set of configurable parameters that are used in training and testing the classifiers. Each issue file - physically a `csv` file - is a collection of issues from JIRA described by 4 fields: (`ID`, `CLASSIFIED`, `TYPE`, `SUMMARY`). The configurable parameters are

1. `max_terms_in_dtm` (internal variable: `MAX_TERMS_IN_DTM`) (default = 100) denoting the number of terms to retain in the DTM. `max_terms_in_dtm` may be a fraction or an integer. If it is a fraction, `MAX_TERMS_IN_DTM` is set to `max_terms_in_dtm` times the actual number of terms in the DTM of the training subset; otherwise it specifies the absolute number of terms to be retained in the pruned training DTM,
2. `normalize` (internal variable: `NORMALIZE`) (default=1, meaning, use raw term frequencies) specifying which of the 7 transforms $T1$ to $T7$ is to be used,
3. `cv_fold` (internal variable: `CV_FOLD`) (default = 10) denoting the number of folds for cross-validation

The output consists of (1) values of precision, recall, and F-measure treating `BUG` as the class of interest, (2) average accuracy, and (3) weighted average F-measure.

Listing 1 Running the R script

```
Rscript BugBang.R --infile="./http_client.csv"
     --outfile="./classification_http_client.out"
     --max_terms_in_dtm=0.25 --normalize=3 --cv_fold=10
```

Dividing dataset for cross-validation: We use k-fold cross-validation where k is set to the value specified in `CV_FOLD`. For each of the k folds, the test subset is generated using stratified sampling.

Generation of DTM: Instead of using the training and test subsets directly, we generate a *DTM* for each subset. First, the training subset is preprocessed and converted to a DTM (or p-training subset). The most frequent terms, counting to no more than `MAX_TERMS_IN_DTM`, are retained in the DTM. The term frequencies in the DTM are normalized according to the value of the `NORMALIZE` parameter. Then the test subset is converted to a DTM (or p-test subset) comprised of only those terms that are already present in the training DTM.

Listing 2 Generation of DTM

```
#Stopwords are listed in stopwords.custom
#Collect functions for preprocessing dataset
skipWords <- function(x) removeWords(x,
     stopwords.custom)
funcs <- list(content_transformer(tolower),
     removePunctuation, removeNumbers, skipWords,
     stripWhitespace, stemDocument);
#Input data has 4 columns (ID, CLASSIFIED, TYPE,
     SUMMARY)
#Training subset is train.in, test subset is test.in
##Create DTM and p-training subset (train.pdataset)
train.corpus   <-
     Corpus(VectorSource(train.in$SUMMARY));
train.corpus   <- tm_map(train.corpus, FUN=tm_reduce,
     tmFuns=funcs);
train.dtm.m    <-
     as.matrix(DocumentTermMatrix(train.corpus));
#Transform raw frequencies to suitable scores
train.dtm.m    <- tfNormalize(train.dtm.m, NORMALIZE);
#Prune DTM
train.dtm.m <- keepMostFrequentTerms(train.dtm.m,
     MAX_TERMS_IN_DTM);
```

```
classified      <- as.factor(train.in$CLASSIFIED);
train.pdataset <- data.frame(classified, train.dtm.m);
#Create dictionary of terms in train.dtm
train.dictionary <- colnames(train.dtm.m);
##Create dtm and p-test subset (test.pdataset)
test.corpus     <-
     Corpus(VectorSource(test.in$SUMMARY));
test.corpus     <- tm_map(test.corpus, FUN=tm_reduce,
     tmFuns=funcs);
test.dtm.full.m <-
     tfNormalize(as.matrix(DocumentTermMatrix(test.corpus)),
     NORMALIZE); #Normalization needs information about
     all terms
test.dtm.m      <-
     as.matrix(DocumentTermMatrix(test.corpus, control
     = list(dictionary=train.dictionary)));
#Now replace the common columns in test.dtm.m with
     normalized columns in test.dtm.full.m
test.dtm.m      <- replaceMatchedColumns(test.dtm.m,
     test.dtm.full.m);
classified      <- as.factor(test.in$CLASSIFIED);
test.pdataset   <- data.frame(classified, test.dtm.m);
```

Classifiers used: We mention in Table 2 the classifiers used, their R implementations and the hyperparameters that we set manually. For hyperparameters not mentioned in the table, default values are used. We did not tune them extensively but did perform some limited experiments on a smaller subset of the data to arrive at the given configurations. For example, in case of `svm`, we carried out a grid search on a subset of the data to infer the 'optimal' values of `cost` and `gamma`.

Table 2 Classifiers used from R

Classifier	R function {package}	Hyperparameters
NB	`naiveBayes {e1071}`	-
LDA	`lda {MASS}`	-
kNN	`knn {class}`	`k=15`
SVM.linear	`svm {e1071}` `(kernel=linear)`	`cost=100`
SVM.rbf	`svm {e1071}` `(kernel=rbf)`	`cost=100,` `gamma=1e-04`
SVM.poly2	`svm {e1071}` `(kernel=polynomial)`	`degree=2,` `cost=100,` `gamma=1e-02`
SVM.poly3	`svm {e1071}` `(kernel=polynomial)`	`degree=3,` `cost=100,` `gamma=1e-02`
SVM.sigmoid	`svm {e1071}` `(kernel=sigmoid)`	`cost=100,` `gamma=1e-04`
DT	`rpart {rpart}`	-
RF	`randomForest` `{randomForest}`	-

Training and testing: In the training phase, the p-training subset is fed into a classifier to build a classification model. Subsequently, we input the p-test subset to check how correctly the trained classifier can label the issues.

Listing 3 Training and testing classifiers

```
#Column 1 of train.pdataset and test.pdataset
#holds true type (CLASSIFIED) of an issue

#kNN classifier
knn.predicted <- knn(train=train.pdataset[,-1],
     test=test.pdataset[,-1], cl=train.pdataset[,1],
     k=15);

#SVM classifier with rbf (Gaussian) kernel
svm.rbf.model <- svm(Type~., data=train.pdataset,
     kernel="radial", gamma=100, cost=1e-04);
svm.rbf.predicted <- predict(svm.rbf.model,
     test.pdataset[,-1]);
```

Iterations: The sequence of generating p-training and p-test subsets, training the classifiers and testing them is iterated `CV_FOLD` times, each time using different training and test subsets. Finally, we analyze the classifiers' performance using standard measures of precision, recall, F-measure, weighted average F-measure and average accuracy. The whole procedure is executed for each project separately by specifying the appropriate input file name.

Listing 4 Overall iteration structure

```
input.dataset <- read.csv(file=INPUT_FILE, header=TRUE,
     sep=",", stringsAsFactors=FALSE);
#Divide input.dataset into CV_FOLD number of
#training and test subsets using stratified sampling
#Call them train.in[[1:CV_FOLD]], test.in[[1:CV_FOLD]]
for(j in 1:CV_FOLD) {
   #Convert train.in[[j]] to train.pdataset
   #Convert test.in[[j]] to test.pdataset
   for(each classifier) {
      #Train with train.pdataset
      #Test with test.pdataset
   }
}
#Output performance results for each classifier
```

6 Experiments, Results and Discussion

6.1 Experiments and Result Analysis

We conduct various experiments to determine how well a classifier can label the instances (for a given classifier configuration) and how various input parameters affect the classifier performance. The selected datasets and the distribution of correct labels in them are shown in Table 3. Note that although the original dataset in [14] contains 2442 entries for Lucene and 2402 entries for Jackrabbit, two entries, namely `LUCENE-3233` (whose both true and Jira types are `NUG`) and `JCR-1530` (whose both true and Jira types are `BUG`), are excluded by our script as the R function `content_transformer(tolower)`

called during preprocessing cannot handle multibyte string present in their SUMMARY. We did not address this encoding issue since the datasets remain appreciably large even after the exclusions.

Table 3 Selected issue reports

Project	Total reports	BUG	NUG
HttpClient	745	305	440
Lucene	2441	697	1744
Jackrabbit	2401	937	1464

Experiment 1. Finding the best classifier: Among the classifiers we used, we intend to find out which one works best on the given datasets. So we train and test each classifier in turn and for each classifier vary the DTM size and the normalization value over a predefined range. The number of terms in a training DTM after preprocessing is typically around 1230 for HttpClient, 2990 for Lucene and 2720 for Jackrabbit. We configure the training DTM size to hold the most frequent 10%, 20%, 25%, 30%, 40%, 50%, 60%, 70% and 75% of all the generated terms. For each DTM size, we vary normalization values from $T1$ through $T7$.

We find that random forest (RF) gives the highest F-measure, average accuracy and weighted average F-measure for any dataset over all DTM sizes and normalization values we considered. Let us take F-measure as an example. The highest F-measure we observed for HttpClient over all DTM sizes and normalization values is 0.687; it is achieved by RF when DTM size 0.25 (i.e., only 25% terms are retained) and normalization value $T3$ are used. Graphs in Figures 6 - 14 show the variation of classifier performance with DTM size, using `normalize=3`, for the 3 projects. Given this normalization value and the DTM sizes considered, for the projects HttpClient, Lucene and Jackrabbit, the highest F-measure values are 0.687, 0.708 and 0.759 respectively, the highest average accuracy values are 0.748, 0.834 and 0.809 respectively and the highest weighted average F-measure values are 0.746, 0.834 and 0.810 respectively. Since the datasets do not contain an equal fraction of BUG and NUG types, weighted average F-measure is likely to be a better indicator of performance than simple F-measure. From the graphs, we find that SVM with sigmoid and rbf kernels and LDA, to some extent, give performance close to that of RF. We observe that the performance of DT is essentially independent of the DTM size when `max_terms_in_dtm` exceeds 0.2 (for a given normalization value). This means, retaining more than 20% terms in the training DTM is not needed.

In case of the top four classifiers, performance does not vary significantly with the normalization scheme. As an example, consider the graph in Figure 15; the F-scores achieved by RF in case of Lucene for DTM size 0.25 and different normalization values do not vary much. From the experimental results, we did not find any particular normalization scheme consistently performing best across all DTM sizes for these top classifiers. However, we observe that kNN performance (for all three measures) is usually better when `normalize=7` is used for a given DTM size. This is probably because `knn` uses Euclidean distance between the instances to determine the class and this normalization scheme scales the feature values effectively for comparing distances.

We feel that using RF classifier with `max_terms_in_dtm` set to a value within 0.2 to 0.3 and `normalize=3` should give very satisfactory results. This means only the most frequent 20%−30% terms along with logarithmic damping of term frequencies are sufficient for high-quality automated classification of issue reports.

Experiment 2. Effect of misclassification / missing labels: We want to examine the need for manually classified reports on the quality of automated classification. So in this experiment, we use the TYPE column instead of the CLASSIFIED column for the training instances. Thus the classifier learns the type assigned in Jira rather than the manually assigned type. There are also many rows in which no label is present in the TYPE field. These rows cannot be used for training. To handle this situation, we ignore these rows if they fall in the training subset. To assess classifier performance, we use the CLASSIFIED column as the ground truth and compare it with the generated label. Table 4 shows the counts of missing labels and incorrect types in the datasets. Column 4 of the table mentions the total number of misclassifications along with the number of cases in which a BUG has been incorrectly labeled as NUG (indicated as B2N) and the number of cases in which a NUG has been incorrectly labeled as BUG (indicated as N2B). Column 5 shows the total number of errors, both as an absolute count and as a percentage of the total number of issue reports in the corresponding project. It is clear from the table that most of the misclassifications involve marking an issue as a BUG when it does not involve corrective maintenance at all. Though not shown in the table, most of the rows with missing labels were also found to be NUG by manual review [14].

We find that use of misclassified reports results in reduced performance of the classifiers for each of the projects. The histogram in Figure 16 shows the F-measure

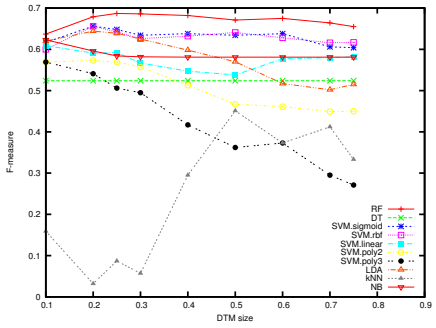

Fig. 6 F-measure for HttpClient project with `normalize=3`.

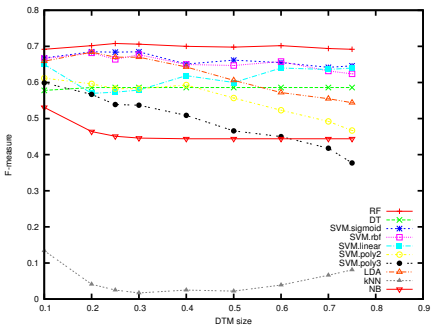

Fig. 7 F-measure for Lucene project with `normalize=3`.

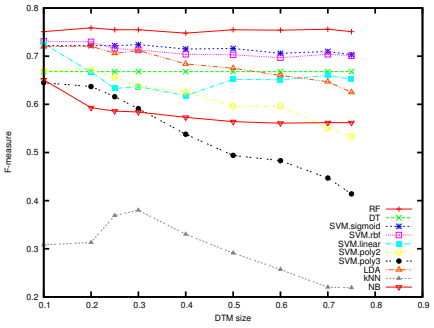

Fig. 8 F-measure for Jackrabbit project with `normalize=3`.

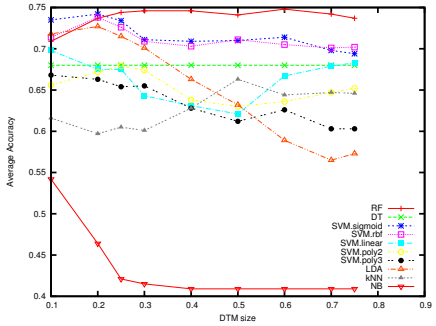

Fig. 9 Average accuracy for HttpClient project with `normalize=3`.

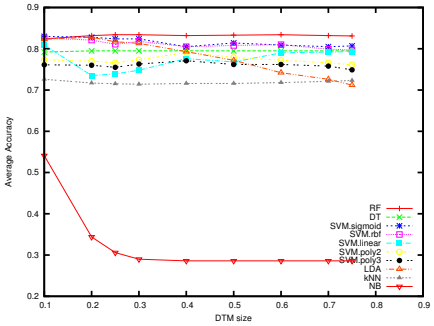

Fig. 10 Average accuracy for Lucene project with `normalize=3`.

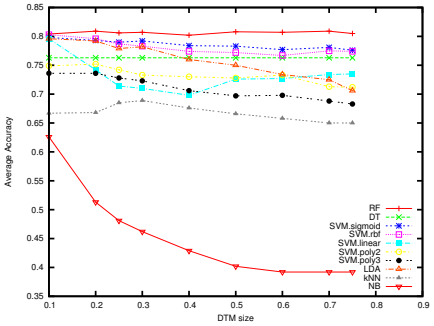

Fig. 11 Average accuracy for Jackrabbit project with `normalize=3`.

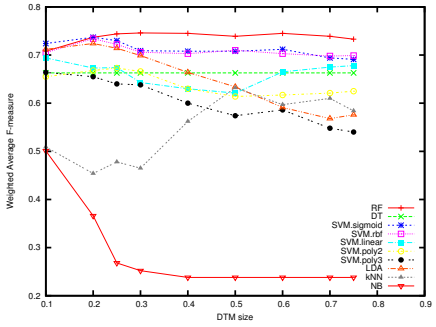

Fig. 12 Weighted average F-measure for HttpClient project with `normalize=3`.

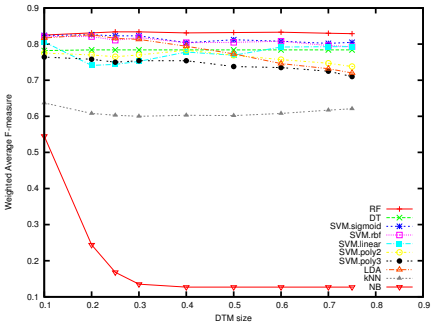

Fig. 13 Weighted average F-measure for Lucene project with `normalize=3`.

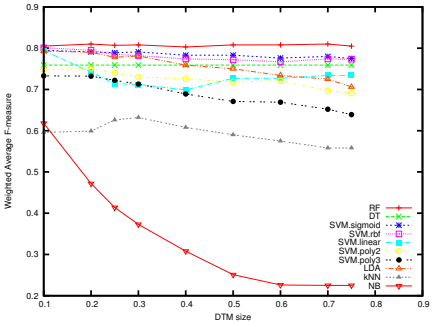

Fig. 14 Weighted average F-measure for Jackrabbit project with `normalize=3`.

Table 4 Errors in issue reports

Project	Total reports	Missing labels	Misclassified reports (B2N, N2B)	Total errors (% of total reports)
HttpClient	745	14	177 (6, 171)	191 (26%)
Lucene	2441	74	328 (26, 302)	402 (16%)
Jackrabbit	2401	61	377 (19, 358)	438 (18%)

(marked as F), average accuracy (marked as Acc) and weighted average F-measure (marked as wt-avg-F) of RF classifier in the two cases, namely when manually corrected issue types are used (e.g., HttpClientCorrect) and when issue types extracted from Jira are used (e.g., HttpClientWithError), for all the 3 projects. We have used `normalize=3` and `max_terms_in_dtm=0.25` in all cases. Similar reduction in performance is also observed for other classifiers and other normalization values and DTM sizes. This underscores the importance of using manually corrected datasets for training.

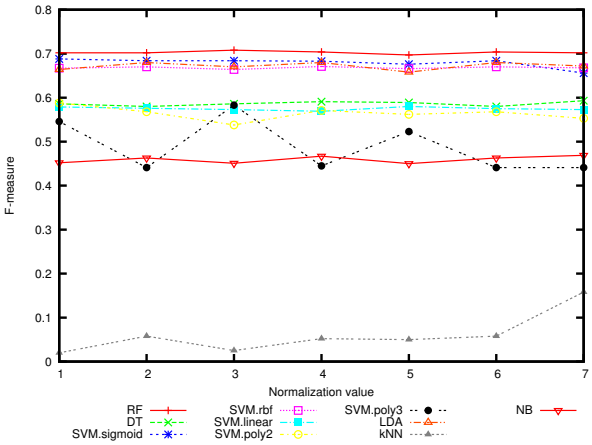

Fig. 15 F-measure for for various normalization values in Lucene project. We have used `max_terms_in_dtm=0.25` in all cases.

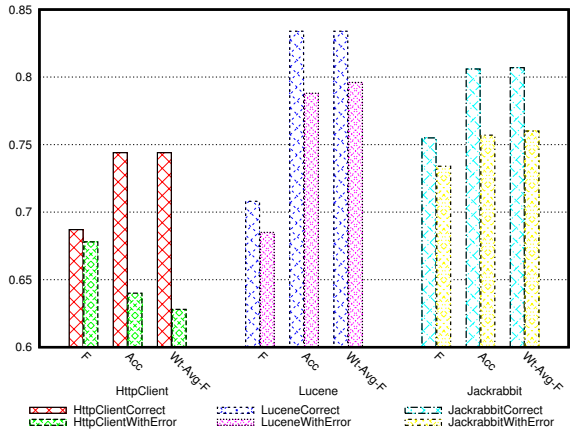

Fig. 16 Comparison of RF performance for the 2 cases, namely when using manually identified true types and when using types present in Jira, for the 3 projects. We have used `normalize=3` and `max_terms_in_dtm=0.25` in all cases.

6.2 Threats to validity

Threats to internal validity: We assume the labels created by [14] as ground truth and train the classifiers accordingly. Hence any errors therein affect the labels generated for the test cases.

Threats to external validity: The results obtained in this paper are sensitive to the choice of datasets. In particular, if the datasets are changed, the outcomes may alter, limiting the generalization of our results. We use classifiers available in an R environment and hence the results may change if other implementations are used. The parameters used to prepare the DTM and tune the classifiers also influence the results.

7 Related Work

Analysis of software issue reports submitted to issue tracking tools is an active area of research due to its applications in triaging bug reports (i.e., the task of finding the most appropriate developer to fix a bug), categorizing bugs into different severity types, estimating issue resolution time, and providing feedback on the quality of reports.

Bug/non-bug binary classification: Antoniol *et al.* [2] manually classified 1800 issues collected from issue trackers of Mozilla, Eclipse and JBoss projects into two classes: bug and non-bug. They investigate the use of various information contained in the issue reports on the classification performance. They also perform automatic classification of (a smaller subset of) issue reports using naive Bayes, ADTree and linear logistic regression classifier. Zhou *et al.* [40] employ text mining followed by data mining techniques to classify bug reports and achieve significant improvement in accuracy and F-measure on the same datasets as used in [2]. The extent and cost of misclassification of software issues are deeply studied in [14]; it reports that 90 working days were needed to manually review 7401 closed and fixed bug reports from 5 open-source projects and that about one-third of them were actually misclassified. This emphaizes the pressing need to invent methods to automatically classify bug reports if possible. Then, even if the original issue type tagged by the reporter is incorrect, the right type can be inferred and used for further analysis by application engineers. Pingclasai *et al.* [28] report results of automated bug report classification on three of these manually sanitized datasets [14]. In particular, they use the datasets from HttpClient, Lucene and Jackrabbit projects. They apply topic modelling (using latent Dirichlet allocation (LDAL[12])) followed by one of the three techniques - ADTree, naive Bayes classifier and logistic regression. Chawla and Singh [9] propose a fuzzy logic based technique to classify issue reports automatically. They report higher values of F-score compared to [28] for each of the preceding three projects. We also use the same datasets as used in [28, 9].

Multiclass bug classification: Closely associated with the above binary classification problem is the issue of grouping bugs into different categories. Recently Ohira *et al.* [26] manually reviewed 4000 issue reports in Jira from four open-source projects and manually classified

[12] Recollect we abbreviate linear discriminant analysis as LDA. So we use LDAL here.

them based on their impact (e.g., security bug, performance bug, etc.) on the project. Limsettho *et al.* [22] use clustering, an unsupervised learning approach, to automatically group bug reports based on their textual similarity. Thung *et al.* [34] apply supervised learning that exploits textual content of bug reports as well as code features to label defects with one of the three defect categories, namely control and data flow, structural, and non-functional. It can be used only for post-mortem analysis of defects since code from bug fixes are also taken as input. LDAL is a popular technique to identify topics in a corpus and is hence useful in categorizing bug reports. Nagwani *et al.* [25] use LDAL to generate the most frequent discriminant terms that describe the bugs in different software bug repositories. Lucas *et al.* [20] collect a number of reports related to NASA missions and use LDAL to identify the most popular topics within and across missions. Chawla and Singh [8] consider the problem of missing labels (which identify the defect category, e.g., security, crash, etc.) in bug reports and propose an automated technique for labeling using tf-idf and latent semantic indexing. Zibran [41] uses labeled LDAL to classify bug reports into a set of 22 predefined categories of API usage factors like concurrency, memory management, etc. Somasundaram and Murphy [31] show that, given a fixed number of components in a software, LDAL combined with Kullback-Leibler divergence algorithm can categorize bug reports more consistently than SVM (with tf-idf and LDAL) across the components even if the number of reports in each category varies widely in the repository. Since parameter tuning in LDAL is an involved exercise, Limsettho *et al.* [21] propose a non-parametric approach, namely hierarchical Dirichlet process, to automatically classify bug reports into multiple classes. It gives performance comparable to LDAL. A review on topic modeling of software repositories appears in [10]. Kochhar *et al.* [18] employ classifiers to automatically check if an issue needs to be *reclassified* and if so, into which of a set of 13 categories. Different from our work, it uses the existing issue category as a feature for reclassification. Nowadays there is a proliferation of mobile apps, often downloaded from Google Play or Apple AppStore. Maalej *et al.* [23] use supervised learning to classify app reviews into four types: bug reports, feature requests, user experiences, and text ratings; multiple binary classifiers are found better than a single multiclass classifier.

Other bug mining problems: The related problem of automated bug triaging that attempts to assign bugs to developers using machine learning techniques has also received significant attention in the literature. Existing studies have used naive Bayes classifier trained on the text of bug reports [24], SVM [4, 3], different classifiers with bug-tossing graphs (that model reassignment of developers to the same bug) [15, 5], ensemble classifiers [16] and fuzzy sets [33]. Researchers have also used machine learning to detect duplicate bug reports (e.g., [1]), identify bug severities (e.g., [19]), predict bug priorities (e.g., [35, 36]), estimate bug fixing time (e.g., [39]), generate feedback on the quality of issue reports (e.g., [42]) and analyze bug report trends [38]. These works belong to the more general theme of mining software repositories to extract valuable information on the quality of a software or its management and offer recommendations [32, 12, 13, 7].

Our perspective: Like the preceding works, we too study automatic binary classification of issue reports on the datasets used by [28] but use a partially different collection of classifiers (e.g., LDA, kNN, SVM and RF are added). Although the F-measures we obtained are lower than those in [28], we do not apply topic modeling that is more involved than our approach since it uses both `Summary` and `Description` and requires complex parameter tuning [22]. Even higher F-measure values are obtained by the fuzzy classification scheme of [9]. In contrast, we perform simple preprocessing of the `Summary` part of a report to generate a document term matrix that is then input to the classifiers. In fact, the best performance values we obtain are comparable to the best ones in [2] that also uses simple text mining, albeit on different projects, features and classifiers. One of our goals is to understand the efficiency of machine learning techniques when minimal preprocessing is applied to the input datasets. Expensive preprocessing is likely to inhibit the inclusion of these tools in real issue trackers that are frequently updated with new reports. We also attempt to identify the classifiers that give the best performance.

8 Conclusion

We used machine learning techniques to automatically classify issue reports into bug and non-bug categories. The experiments were done using issue reports of three large open-source projects. Our results indicate that classifiers like random forest and SVM can achieve accuracy of $75 - 83\%$ depending on the project. This is interesting given that we perform minimal preprocessing on the data and run our experiments for three different projects. Our results suggest it might be useful to classify issue reports using simple classifiers before further analysis by developers. While constructing the training DTM, we chose the most frequent terms across

bug and non-bug categories. In future, we would like to construct separate dictionaries for the two categories. We also aim develop a plugin for online issue classification in issue trackers. Other research directions include exploring whether more attributes from the reports lead to better classification, tuning the classifiers to a greater degree, expanding the datasets and classifiers and determining how to select a training subset in practice when faced with a very large repository of issue reports.

A Tables on Classifier Performance

This appendix shows the tables used to plot the graphs in section 6 of the paper. The highest performance value in a row is highlighted with red color in Tables 5 through 14. The highest performance value for every project in a row is highlighted with red color in Table 15

Table 5 F-measure for HttpClient with `normalize=3` (cf. Figure 6)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.623	0.620	0.159	0.609	0.600	0.569	0.569	0.617	0.524	0.637
0.2	0.596	0.644	0.032	0.591	0.656	0.573	0.541	0.656	0.524	0.679
0.25	0.584	0.639	0.087	0.592	0.642	0.569	0.506	0.648	0.524	0.687
0.3	0.582	0.625	0.057	0.567	0.626	0.557	0.495	0.634	0.524	0.686
0.4	0.581	0.598	0.295	0.547	0.632	0.513	0.417	0.638	0.524	0.682
0.5	0.581	0.570	0.451	0.538	0.641	0.467	0.362	0.634	0.524	0.671
0.6	0.581	0.517	0.373	0.577	0.628	0.461	0.373	0.638	0.524	0.675
0.7	0.581	0.502	0.412	0.580	0.616	0.449	0.295	0.606	0.524	0.664
0.75	0.581	0.515	0.333	0.582	0.617	0.450	0.271	0.604	0.524	0.655

Table 6 F-measure for Lucene with `normalize=3` (cf. Figure 7)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.531	0.659	0.134	0.650	0.666	0.612	0.599	0.668	0.578	0.692
0.2	0.464	0.684	0.042	0.570	0.682	0.596	0.567	0.685	0.586	0.702
0.25	0.451	0.670	0.025	0.573	0.664	0.583	0.539	0.684	0.586	0.708
0.3	0.446	0.670	0.017	0.579	0.677	0.583	0.537	0.685	0.586	0.706
0.4	0.444	0.643	0.025	0.619	0.650	0.593	0.509	0.651	0.586	0.700
0.5	0.444	0.606	0.023	0.600	0.647	0.557	0.466	0.662	0.586	0.698
0.6	0.444	0.572	0.039	0.640	0.658	0.523	0.450	0.655	0.586	0.702
0.7	0.444	0.555	0.066	0.636	0.632	0.492	0.418	0.642	0.586	0.694
0.75	0.444	0.544	0.081	0.640	0.624	0.467	0.377	0.646	0.586	0.692

Table 7 F-measure for Jackrabbit with `normalize=3` (cf. Figure 8)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.651	0.720	0.308	0.726	0.731	0.670	0.645	0.722	0.668	0.751
0.2	0.593	0.721	0.313	0.667	0.730	0.672	0.637	0.722	0.668	0.759
0.25	0.586	0.706	0.369	0.633	0.716	0.656	0.616	0.722	0.668	0.755
0.3	0.584	0.712	0.380	0.637	0.713	0.638	0.591	0.724	0.668	0.755
0.4	0.573	0.684	0.330	0.618	0.704	0.626	0.538	0.715	0.668	0.748
0.5	0.564	0.675	0.291	0.652	0.703	0.596	0.494	0.716	0.668	0.755
0.6	0.561	0.660	0.257	0.651	0.697	0.596	0.483	0.706	0.668	0.754
0.7	0.562	0.647	0.220	0.660	0.704	0.551	0.447	0.710	0.668	0.756
0.75	0.562	0.625	0.219	0.652	0.701	0.533	0.414	0.703	0.668	0.751

Table 8 Average accuracy for HttpClient with `normalize=3` (cf. Figure 9)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.542	0.718	0.616	0.698	0.714	0.656	0.668	0.735	0.680	0.709
0.2	0.464	0.727	0.597	0.675	0.738	0.672	0.663	0.742	0.680	0.737
0.25	0.421	0.715	0.605	0.675	0.726	0.679	0.654	0.734	0.680	0.744
0.3	0.415	0.701	0.601	0.643	0.709	0.674	0.655	0.711	0.680	0.746
0.4	0.409	0.663	0.628	0.631	0.703	0.638	0.628	0.709	0.680	0.746
0.5	0.409	0.632	0.663	0.621	0.711	0.630	0.612	0.710	0.680	0.741
0.6	0.409	0.589	0.644	0.667	0.705	0.636	0.626	0.714	0.680	0.748
0.7	0.409	0.565	0.647	0.679	0.701	0.647	0.603	0.698	0.680	0.742
0.75	0.409	0.573	0.646	0.683	0.702	0.652	0.603	0.694	0.680	0.737

Table 9 Average accuracy for Lucene with `normalize=3` (cf. Figure 10)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.541	0.823	0.726	0.809	0.826	0.773	0.761	0.830	0.792	0.824
0.2	0.344	0.828	0.717	0.735	0.821	0.770	0.760	0.828	0.795	0.832
0.25	0.306	0.817	0.715	0.739	0.811	0.766	0.755	0.824	0.795	0.834
0.3	0.290	0.813	0.714	0.748	0.818	0.773	0.763	0.824	0.795	0.834
0.4	0.286	0.793	0.716	0.776	0.805	0.791	0.771	0.805	0.795	0.832
0.5	0.286	0.772	0.716	0.768	0.808	0.779	0.762	0.814	0.795	0.833
0.6	0.286	0.742	0.718	0.790	0.810	0.772	0.762	0.809	0.795	0.834
0.7	0.286	0.726	0.721	0.792	0.799	0.766	0.758	0.805	0.795	0.832
0.75	0.286	0.712	0.723	0.793	0.797	0.761	0.749	0.807	0.795	0.831

Table 10 Average accuracy for Jackrabbit with `normalize=3` (cf. Figure 11)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.626	0.796	0.667	0.796	0.803	0.749	0.736	0.798	0.763	0.804
0.2	0.513	0.792	0.668	0.742	0.796	0.752	0.736	0.793	0.763	0.809
0.25	0.481	0.779	0.685	0.714	0.787	0.742	0.728	0.790	0.763	0.806
0.3	0.462	0.782	0.689	0.710	0.783	0.733	0.723	0.792	0.763	0.807
0.4	0.429	0.760	0.676	0.698	0.774	0.730	0.706	0.784	0.763	0.802
0.5	0.402	0.750	0.666	0.726	0.772	0.728	0.697	0.783	0.763	0.808
0.6	0.392	0.734	0.658	0.727	0.767	0.733	0.698	0.777	0.763	0.807
0.7	0.392	0.725	0.650	0.734	0.775	0.713	0.688	0.781	0.763	0.809
0.75	0.392	0.706	0.650	0.735	0.774	0.712	0.683	0.776	0.763	0.805

Table 11 Weighted average F-Measure for HttpClient with `normalize=3` (cf. Figure 12)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.501	0.712	0.509	0.694	0.705	0.655	0.664	0.724	0.663	0.708
0.2	0.366	0.724	0.454	0.673	0.734	0.668	0.655	0.737	0.663	0.737
0.25	0.268	0.714	0.478	0.674	0.722	0.673	0.640	0.730	0.663	0.744
0.3	0.252	0.699	0.465	0.643	0.706	0.666	0.638	0.709	0.663	0.746
0.4	0.238	0.664	0.562	0.630	0.703	0.630	0.600	0.708	0.663	0.745
0.5	0.238	0.634	0.632	0.621	0.710	0.614	0.574	0.708	0.663	0.739
0.6	0.238	0.591	0.597	0.665	0.703	0.617	0.586	0.712	0.663	0.745
0.7	0.238	0.568	0.610	0.675	0.698	0.621	0.548	0.694	0.663	0.739
0.75	0.238	0.576	0.584	0.678	0.699	0.625	0.540	0.691	0.663	0.733

Table 12 Weighted average F-Measure for Lucene with `normalize=3` (cf. Figure 13)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.545	0.817	0.637	0.806	0.821	0.775	0.764	0.824	0.782	0.825
0.2	0.244	0.826	0.608	0.741	0.821	0.770	0.758	0.826	0.784	0.831
0.25	0.168	0.816	0.603	0.744	0.811	0.765	0.750	0.823	0.784	0.834
0.3	0.135	0.813	0.600	0.752	0.818	0.770	0.754	0.823	0.784	0.834
0.4	0.127	0.794	0.603	0.778	0.804	0.783	0.754	0.804	0.784	0.831
0.5	0.127	0.773	0.602	0.769	0.805	0.769	0.738	0.812	0.784	0.832
0.6	0.127	0.746	0.608	0.792	0.808	0.757	0.735	0.808	0.784	0.833
0.7	0.127	0.732	0.617	0.793	0.796	0.747	0.725	0.802	0.784	0.830
0.75	0.127	0.720	0.621	0.793	0.793	0.738	0.710	0.805	0.784	0.829

Table 13 Weighted average F-Measure for Jackrabbit with `normalize=3` (cf. Figure 14)

DTM size	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
0.1	0.618	0.793	0.596	0.794	0.801	0.748	0.733	0.795	0.759	0.805
0.2	0.472	0.790	0.599	0.742	0.795	0.751	0.732	0.791	0.759	0.810
0.25	0.414	0.778	0.626	0.714	0.785	0.740	0.722	0.789	0.759	0.807
0.3	0.373	0.781	0.632	0.712	0.782	0.730	0.713	0.791	0.759	0.808
0.4	0.308	0.759	0.608	0.699	0.774	0.726	0.689	0.783	0.759	0.803
0.5	0.251	0.750	0.590	0.727	0.772	0.718	0.671	0.783	0.759	0.808
0.6	0.226	0.734	0.575	0.727	0.767	0.721	0.669	0.776	0.759	0.808
0.7	0.225	0.725	0.558	0.734	0.774	0.697	0.652	0.780	0.759	0.810
0.75	0.225	0.706	0.558	0.734	0.773	0.691	0.639	0.775	0.759	0.805

Table 14 F-measures for Lucene with `max_terms_in_dtm=0.25` (cf. Figure 15)

Norm	NB	LDA	kNN	SVM.linear	SVM.rbf	SVM.poly2	SVM.poly3	SVM.sigmoid	DT	RF
1	0.452	0.664	0.020	0.579	0.667	0.588	0.546	0.688	0.586	0.702
2	0.463	0.680	0.058	0.576	0.670	0.568	0.441	0.684	0.580	0.702
3	0.451	0.670	0.025	0.573	0.664	0.538	0.583	0.684	0.586	0.708
4	0.467	0.680	0.052	0.569	0.671	0.571	0.445	0.683	0.591	0.704
5	0.450	0.658	0.050	0.580	0.666	0.562	0.523	0.676	0.589	0.697
6	0.463	0.680	0.058	0.575	0.670	0.568	0.441	0.684	0.580	0.704
7	0.469	0.672	0.158	0.573	0.668	0.553	0.441	0.656	0.593	0.702

Table 15 Performance of random forest (RF) with `normalize=3` and `max_terms_in_dtm=0.25` on manually classified reports (denoted by *Correct) vis-a-vis reports with types mentioned in Jira (denoted by *WithError) (cf. Figure 16)

Measure	HttpClientCorrect	HttpClientWithError	LuceneCorrect	LuceneWithError	JackrabbitCorrect	JackrabbitWithError
F	0.687	0.678	0.708	0.685	0.755	0.734
Acc	0.744	0.640	0.834	0.788	0.806	0.757
Wt-Avg-F	0.744	0.628	0.834	0.796	0.807	0.760

Acknowledgements We are immensely thankful to Professor Hideaki Hata, Nara Institute of Science and Technology, Nara, Japan for providing us the datasets used in [28] (and based on [14]) which helped us a lot in generating the datasets used in this paper. We also express our thanks to Ananda Das, Research Scholar, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur for helping us set up the R software environment.

References

1. Aggarwal K, Timbers F, Rutgers T, Hindle A, Stroulia E, Greiner R (2016) Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process
2. 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, pp 23:304–23:318.
3. Anvik J, Murphy GC (2011) Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology 20(3):10
4. Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), ACM, pp 361–370
5. Bhattacharya P, Neamtiu I, Shelton CR (2012) Automated, highly-accurate, bug assignment using machine learning and tossing graphs. Journal of Systems and Software 85(10):2275–2292
6. Breiman L (2001) Random forests. Machine learning 45(1):5–32
7. Cavalcanti YC, Mota Silveira Neto PA, Machado IdC, Vale TF, Almeida ES, Meira SRdL (2014) Challenges and opportunities for software change request repositories: a systematic mapping study. Journal of Software: Evolution and Process 26(7):620–653
8. Chawla I, Singh SK (2014) Automatic bug labeling using semantic information from lsi. In: Proceedings of the 2014 7th International Conference on Contemporary Computing (IC3'14), IEEE, pp 376–381
9. Chawla I, Singh SK (2015) An automated approach for bug categorization using fuzzy logic. In: Proceedings of the 8th India Software Engineering Conference (ISEC'15), ACM, pp 90–99
10. Chen TH, Thomas SW, Hassan AE (2016) A survey on the use of topic models when mining software repositories. Empirical Software Engineering 21(5):1843–1919
11. Forman G, Scholz M (2010) Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. ACM SIGKDD Explorations Newsletter 12(1):49–57
12. Hemmati H, Nadi S, Baysal O, Kononenko O, Wang W, Holmes R, Godfrey MW (2013) The msr cookbook: Mining a decade of research. In: Proceedings of the 2013 10th IEEE Working Conference on Mining Software Repositories (MSR'13), IEEE, pp 343–352
13. Herzig K, Zeller A (2014) Mining bug data. In: Recommendation Systems in Software Engineering, Springer, pp 131–171
14. 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), IEEE, pp 392–401
15. Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, pp 111–120
16. Jonsson L, Borg M, Broman D, Sandahl K, Eldh S, Runeson P (2016) Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empirical Software Engineering 21(4):1533–1578
17. Ko A, Myers B, Chau D (2006) A linguistic analysis of how people describe software problems. In: Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06), IEEE, pp 127–134
18. Kochhar PS, Thung F, Lo D (2014) Automatic fine-grained issue report reclassification. In: Proceedings of the 2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS'14), IEEE, pp 126–135
19. Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR'10), IEEE, pp 1–10
20. Layman L, Nikora AP, Meek J, Menzies T (2016) Topic modeling of nasa space system problem reports: research in practice. In: Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16), ACM, pp 303–314
21. Limsettho N, Hata H, Matsumoto Ki (2014) Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification. In: Proceedings of the 2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'14), IEEE, pp 1–6
22. Limsettho N, Hata H, Monden A, Matsumoto K (2014) Automatic unsupervised bug report categorization. In: Proceedings of the 2014 6th International Workshop on Empirical Software Engineering in Practice (IWESEP'14), IEEE, pp 7–12
23. Maalej W, Kurtanović Z, Nabil H, Stanik C (2016) On the automatic classification of app reviews. Requirements Engineering 21(3):311–331
24. Murphy G, Čubranić D (2004) Automatic bug triage using text categorization. In: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE'04)
25. Nagwani N, Verma S, Mehta KK (2013) Generating taxonomic terms for software bug classification by utilizing topic models based on latent dirichlet allocation. In: Proceedings of the 2013 11th International Conference on ICT and Knowledge Engineering (ICT&KE'13), IEEE, pp 1–5
26. Ohira M, Kashiwa Y, Yamatani Y, Yoshiyuki H, Maeda Y, Limsettho N, Fujino K, Hata H, Ihara A, Matsumoto K (2015) A dataset of high impact bugs: manually-classified issue reports. In: Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR'15), IEEE, pp 518–521
27. Pandey N, Hudait A, Sanyal DK, Sen A (2016) Automated classification of issue reports from a software issue tracker. In: Proceedings of the 4th International Conference on Advanced Computing, Networking, and Informatics (ICACNI'16), Springer

28. Pingclasai N, Hata H, Matsumoto Ki (2013) Classifying bug reports to bugs and other requests using topic modeling. In: Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC'13), IEEE, vol 2, pp 13–18
29. Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Information processing & management 24(5):513–523
30. Sebastiani F (2002) Machine learning in automated text categorization. ACM Computing Surveys 34(1):1–47
31. Somasundaram K, Murphy GC (2012) Automatic categorization of bug reports using latent dirichlet allocation. In: Proceedings of the 5th India Software Engineering Conference (ISEC'15), ACM, pp 125–130
32. Strate JD, Laplante PA (2013) A literature review of research in software defect reporting. IEEE Transactions on Reliability 62(2):444–454
33. Tamrawi A, Nguyen TT, Al-Kofahi J, Nguyen TN (2011) Fuzzy set-based automatic bug triaging. In: Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE'13), IEEE, pp 884–887
34. Thung F, Lo D, Jiang L (2012) Automatic defect categorization. In: Proceedings of the 2012 19th Working Conference on Reverse Engineering (WCRE'12), IEEE, pp 205–214
35. Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. Empirical Software Engineering 20(5):1354–1383
36. Uddin J, Ghazali R, Deris MM, Naseem R, Shah H (2016) A survey on bug prioritization. Artificial Intelligence Review pp 1–36
37. Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, pp 461–470
38. Wu L, Boyi X, Kaiser G, Passonneau R (2011) Bugminer: software reliability analysis via data mining of bug reports. In: Proceedings of the 23rd Internal Conference on Software Engineering and Knowledge Engineering (SEKE'11), pp 95–100.
39. Zhang H, Gong L, Versteeg S (2013) Predicting bugfixing time: an empirical study of commercial software projects. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE'13), IEEE, pp 1042–1051
40. Zhou Y, Tong Y, Gu R, Gall H (2016) Combining text mining and data mining for bug report classification. Journal of Software: Evolution and Process 28(3):150–176
41. Zibran MF (2016) On the effectiveness of labeled latent dirichlet allocation in automatic bug-report categorization. In: Proceedings of the 38th International Conference on Software Engineering (ICSE'16) Companion, ACM, pp 713–715
42. Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? IEEE Transactions on Software Engineering 36(5):618–643