Combining Text Mining and Data Mining for Bug Report Classification

Yu Zhou[1,2], Yanxiang Tong[1], Ruihang Gu[1], Harald Gall[3]
[1]College of Computer Science, Nanjing University of Aeronautics and Astronautics
[2]State Key Lab. of Novel Software Technology, Nanjing University, China
[3]Department of Informatics, University of Zurich, Switzerland
Email: {zhouyu,tyx,grh}@nuaa.edu.cn, gall@ifi.uzh.ch

***Abstract*—Misclassification of bug reports inevitably sacrifices the performance of bug prediction models. Manual examinations can help reduce the noise but bring a heavy burden for developers instead. In this paper, we propose a hybrid approach by combining both text mining and data mining techniques of bug report data to automate the prediction process. The first stage leverages text mining techniques to analyze the summary parts of bug reports and classifies them into three levels of probability. The extracted features and some other structured features of bug reports are then fed into the machine learner in the second stage. Data grafting techniques are employed to bridge the two stages. Comparative experiments with previous studies on the same data—three large-scale open source projects—consistently achieve a reasonable enhancement (from 77.4% to 81.7%, 73.9% to 80.2% and 87.4% to 93.7%, respectively) over their best results in terms of overall performance. Additional comparative empirical experiments on other two popular open source repositories confirm the findings and demonstrate the benefits of our approach.**

I. Introduction

Recent years have witnessed a booming interest in predicting various aspects of software defects—such as number, locations, or severity—based on the data collected from repositories [1], [2], [3], [4]. With the aim of reliably identifying those defects efficiently, and thus reducing the overall maintenance cost, many statistical models have been developed in the literature [5]. Generally, such models rely on the historical information to make predictions. Among many information sources, bug tracking systems (BTS) are of particular importance due to the richness of defect information as well as the broad application in practice [6].

However, bug reports kept in the BTS, as in their general sense, could also document perfective or adaptive maintenance concerns apart from real (*corrective*) bug descriptions. As observed by Antoniol et al. [7] and others [8], [9], a considerable amount of files marked as defective actually never had a bug (*false positive*). Since an accurate/high-quality data set is always the prerequisite of any effective empirical analysis, the misclassification of bug reports inevitably introduces bias, and thus hampers the prediction performance, making the results doubtful or even misleading [8], [10]. A close examination of the documents so as to identify the real bug reports is not only necessary but also fundamental to the validity of those prediction efforts. Manual classification can help reduce the noise, however, when the number of reports grows large, the associated time and additional costs make it an impractical option. Therefore, an automated mechanism to tell whether or not a given bug report is corrective would be much preferable to augment productivity.

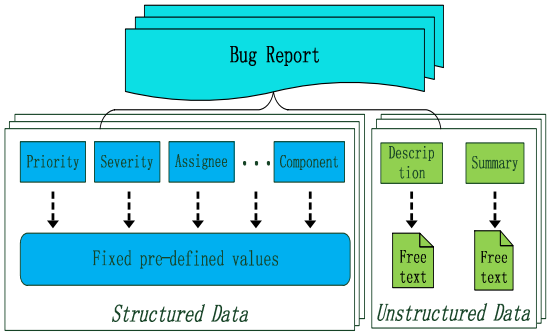

Fig. 1. Bug report fields illustration

Antoniol et al. [7] investigated the automatic classification of bug reports by utilizing conventional text mining techniques, which demonstrated the feasibility. However, the work only considered the description part (free text) of the reports. Typical bug reports are actually *semi-structured* documents consisting of a fixed set of attributes as well as informal narrations. The contents of description (including discussions) and summary parts are indeed unstructured free texts written in natural languages, or even computer-generated stack traces. Nevertheless, the other fields usually have finite or well-defined values, for example, *severity, priority, component, keywords, etc.*, and these fields form the structural aspect of the documents. Therefore, the reports exhibit a mixed characteristic as illustrated in Figure 1.

Surprisingly, very few prior works have formally addressed the implication of incorporating those factors onto the classification results. To some extent, works such as [3], [11], [12], [13], [14] have started to look into the relation of unstructured texts and some selected fields. We argue that this structural information is a valuable asset and its inclusion can contribute to better bug prediction models [15], [16].

In this paper, we want to answer the *question of whether or not a given bug report is a corrective one through a hybrid approach combining both text mining and data mining techniques*. In contrast to purely mining textual descriptions [7], we also consider the structural information with the aim of improving the prediction accuracy. We conduct an extensive empirical study on a collection of more than 3,200 bug reports

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.53

randomly selected from five large open source projects to validate our approach. Overall, our work makes the following contributions:

1) We identify the benefits of incorporating multiple fields of bug reports to predict corrective ones. To the best of our knowledge, this is the first empirical study on combining several different kinds of feature sources besides ordinary textual information (descriptions, discussions, summaries, etc.) from bug reports to predict whether or not they are corrective.
2) We propose a hybrid approach which allows for a composition of models suitable for information sources of different nature. It is essentially open and independent of any specific mining algorithms. Bug reports from five large open source projects are studied, three of which are the same as those in [7], i.e., Mozilla, Eclipse, JBoss. The comparative study confirms the performance improvement by our approach.
3) We use our approach to analyze the bug reports from two other well-known open source projects, i.e., Firefox, OpenFOAM. On one hand, the examination result confirms the findings of existing reports [8], [9] on the presence of a considerable portion of noise; on the other hand, across all the five cases, the experimental result again indicates the feasibility of our approach in terms of performance improvement over the underlying individual classification models.

The remainder of the paper is organized as follows: Section II introduces some background on bug reports and relevant mining techniques. Section III presents a detailed account of our approach. The experiments with performance evaluation are explained in Section IV followed by a discussion in Section V. Related works are discussed in Section VI; Section VII presents our conclusions.

II. Background

Bug reports are managed by BTS and filed by either software testers or end users. A good bug report is supposed to contain detailed information of a bug in order to reproduce it. Generally, bug reports include a fixed set of fields, for example, the date and time when a bug is found, long descriptions and one-line summary of the symptoms, the bug's severity and priority as perceived by the reporter and the product component where the defect occurs etc. When a bug report is submitted, a triager would examine the content and determine whether it is valid and who should work on it. Basically, if a report has been confirmed and verified, its status gets changed to *NEW*. Once a developer has started to work on it, the status gets changed to *ASSIGNED* and finally reaches *RESOLVED* or *CLOSED*. Since reporters usually have different levels of experience and knowledge about the underlying software, the submitted reports are of various quality. As reported in [17], well-written bug reports are likely to get more attention among developers than poorly written ones.

Text classification relates with the process of assigning predefined categories to a document [18]. It can be regarded as a mapping f from a set of categories $C = \{c_1, ..., c_m\}, (|c| \geq 2)$ to a set of documents $D = \{d_1, ..., d_n\}$ resulting in a Boolean decision matrix.

In the value matrix, if more than one value in each column are allowed to be *true*, it's often called *multi-label* classification; while the case in which exactly one value in each column can be assigned to be *true* is called *single-label* classification. A special case in *single-label* classification, in which there are exactly two categories, i.e., $C = \{c_1, c_2\}$, is defined as *binary* classification, and otherwise as *multi-class* classification [19]. In our study, we need to tell whether or not a given bug report is a corrective one, i.e., *bug* or *non-bug*, so it belongs to the *binary* classification problem.

Preprocessing is a critical step in text classification, giving rise to a relative canonical representation of textual descriptions. A typical preprocessing phase usually consists of the following steps: tokenization, stop word removal and stemming. *Tokenization* is the process of splitting a text document into a stream of words by removing all punctuation marks and by replacing tabs and other non-text characters by single white spaces [20]. A collection of words are extracted from the raw documents afterwards. However, many of the frequently used words—such as propositions, conjunctions and articles—are actually used for grammatical soundness, instead of conveying content information and these words are termed as *stop words*. Since the stop words occur too often to contribute to the differentiation across documents, in most cases, they are removed from the set of words obtained after tokenization. *Stemming* maps related words to their same basic form, and thus the indexing size of words can be considerably reduced. For example, the words "modifications," "modifies," "modified," and "modify" can all be reduced to the same root "modif".

Given a set of classifiers, some would yield the better result over the others, but the sets of misclassified patterns of each would not necessarily overlap. This suggests that different classifiers potentially offer complementary information about the patterns to be classified which could be harnessed to improve the performance of the selected classifier [21]. Thus the combination of classifiers can possibly get better performance than the individual. Generally, there are two application scenarios of classifier ensembles. In the first scenario, all the classifiers use the same representation of the input pattern; and in the second, each classifier uses its own representation of the input pattern [21]. Since in our case the bug reports are of mixed characteristics in terms of structuredness, our study fits the second scenario.

III. Approach

Our approach consists of the following steps:

1) *The first stage* uses the summary part of each bug report and classifies it into discrete levels of being a corrective report. The classification is pursued via certain text mining algorithms. In our study, we use *Multinomial Naive Bayes Classifier* and define three levels of possibilities, i.e., $\{high, middle, low\}$. The output levels are regarded as the extracted features from these unstructured texts.
2) *The second stage* uses a sub-set of the structured features of each bug report together with the extracted features from Stage 1. These features and their

concrete values are fed into a machine learner to compute a prediction that can then be analyzed. In our study, we use *Bayesian Net Classifier* as the machine learner.

3) *Data grafting* bridges the two stages. It collects the output of the first text classifier, locates the corresponding bug report archived in the repository, and merges it with other selected features from the same report. The output of data grafting is the input of the machine learner in the second stage.

We use supervised learning in each of the two stages, which means the instances are manually pre-labeled. Each stage is explained in detail below. Figure 2 gives an overview of our approach.

A. Stage 1 – Classifying the summary part

In this stage, the main concern is to classify the textual part of bug reports. Basically, there are two main information sources in the form of free texts: summary and description. In [22], Ko et al. conduct a linguistic analysis on how people describe software problems in bug reports and conclude that one-line summary (title) have almost the same basic content with long descriptions and can be relatively accurately parsed. In a subsequent work, Lamkanfi et al. [3] confirm the findings via an extensive empirical study. Indeed, the summary distills the same information into a one-line sentence by omitting unnecessary details. This difference directly affects automated analysis. Since the description tends to be much longer, the information becomes scattered and more difficult to be extracted. This results in a degradation of performance of automated prediction models relying on it. Therefore, in our approach, we only consider the summary part.

Instead of directly giving the answer of *yes* or *no*, we classify the summaries into three levels of likelihood of being a corrective one, i.e., $\{high, middle, low\}$. The reason is to mimic the manual inspection process. As during the manual examination, we frequently encounter some reports which are rather difficult to tell their category based on the summary per se. For example, in OpenFOAM[1] BTS, the summary of report ID #249 says "solution of reactingFoam is strongly dependent on time step size", and in Firefox[2] BTS, the summary of report ID #330061 says that "Allow only one Places window". It is ambiguous without resorting to other information and thus could be either a feature request/enhancement or a defect in both cases.

To manually classify these summaries, we adopt the heuristics of classification proposed in [8] but with moderate changes. In [8], Herzig et al. list six categories of bug reports, i.e., {*BUG, RFE, IMPR, DOC, REFAC, OTHER*} which clearly distinguish the kind of maintenance work required to resolve the task. Since a fine-grained classification of bug reports is beyond the scope of this paper, we are more interested in a binary classification, i.e., *Bug* and *Nonbug*. Therefore, we group the rest categories enumerated in [8] except for *BUG* into a single category *Nonbug*. In addition, we limit the information domain solely within the *Summary* part in this stage. The first three authors inspected the summaries, and rated their possibility independently. If a summary clearly fits the rule of *Bug* or *Nonbug*, it is labeled as *high* or *low*, respectively. If it is ambiguous or hard to decide, it is labeled as *middle*. To make the final decision and resolve the conflicts, a majority-vote mechanism is used [7]. A rare case is that each inspector assigns a different value to the same summary. Such kind of conflicts are resolved by discussions until a consensus is achieved. Although, these labels are simply meaningless symbols in the sense of text classification, they can be regarded as the confidence index for the final classification.

Preprocessing the summaries includes tokenization, stop-word removal and stemming. Firstly, these summaries are split into sets of words separated by whitespace characters, such as a space/tabular/return, or punctuation (*tokenization*). Secondly, the meaningless stop-words, such as "an" or "the" are removed to reduce the index space (*stop-word removal*). Thirdly, the set of words are again reduced to their root form (*stemming*). We utilize the APIs of *Analyzer* component in Lucene[3] to facilitate the preprocessing. Afterwards, the remaining words are all lowercase.

The obtained terms can then be loaded into a text classifier. Among the many, Naive Bayes is perhaps the most popular model of probabilistic classifiers [23]. The assumption that any two coordinates of the document vector are statistically independent of each other makes it a simplistic yet effective choice in real applications of categorization [20]. A variety of work leverage the model [7], [24], [25], [9], [3], [26]. Without loss of generality, in our approach, we use a Multinomial Naive Bayes Classifier in the first stage.

Based on Multinomial Naive Bayes model, in our case the probability of a summary s being in the category $c \in \{high, middle, low\}$ is computed as:

$$P(c|s) = P(c) * \prod_{1 \leq k \leq n_s} P(t_k|c) \quad (1)$$

$P(t_k|c)$ is the conditional probability of term t_k occurring in a bug report of category c. n_s is the number of meaningful tokens in s and $P(c)$ is the prior probability of an issue report occurring in category c.

For example, in OpenFOAM BTS, the summary of report ID # 71 is "Utility crashes on wedge symmetry axis", the term set might be {*util, crash, wedg, symmetri, axi*} after preprocessing, with $n_s = 5$. We estimate $P(c)$ and $P(t_k|c)$ from the training set and the estimated values are denoted as $\tilde{P}(c)$ and $\tilde{P}(t_k|c)$ respectively. The conditional probability $\tilde{P}(t_k|c)$ can be estimated by the frequency of the term t_k in the summary belonging to category c: $\tilde{P}(t_k|c) = T_{ct}/(\sum_{t' \in V} T_{ct'})$ where T_{ct} is the number of occurrences of t_k in the training set from category c, and multiple occurrences in a summary are counted [27]. Based on these formulae, we can calculate the conditional probabilities of each category given the summary in the above example as follows:

$\tilde{P}(c_k|s) = \tilde{P}(c_k) * \tilde{P}(util|c_k) * \tilde{P}(crash|c_k) * \tilde{P}(wedg|c_k) * \tilde{P}(symmetri|c_k) * \tilde{P}(axi|c_k)$, where $c_k \in \{high, middle, low\}$.

[1] http://www.openfoam.org/

[2] www.mozilla.org/en-US/firefox

[3] https://lucene.apache.org/

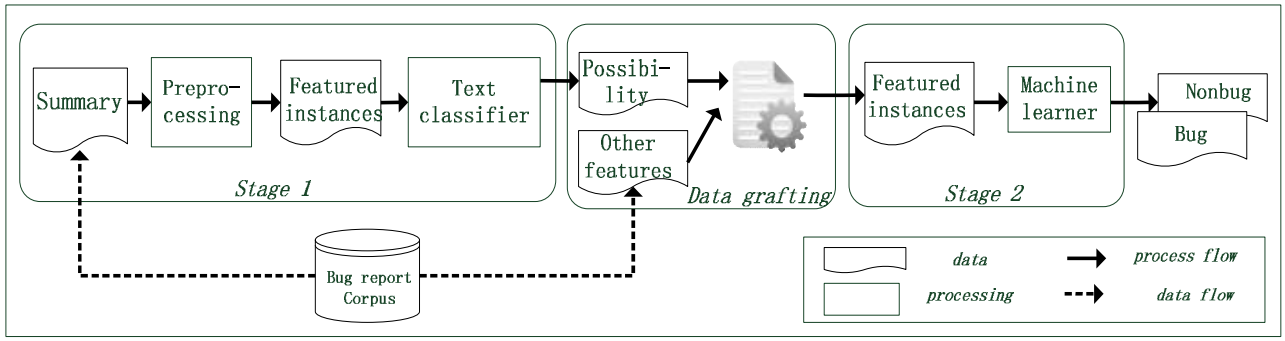

Fig. 2. Approach overview

The classifier assigns the summary s to the category c_k whose result is the highest among the three. We use WEKA [28] tool-set to automate the frequency and probability calculation.

B. Stage 2 – Machine learning

The summary has already been discretized into predefined three levels, together with the other selected fields of bug reports. In Stage 2, the instances can be fed into ordinary machine learners tuned for structured data. Particularly, we use *Bayesian Network Classifier* due to its effectiveness in capturing the underlying causality. Different from the assumption of feature independency made by Naive Bayes Models, Bayesian Network Classifier admits the dependency and expresses the relation via a conditional probability. This can be leveraged to model the feature correlations of bug reports as identified in the literature, such as [11], [29], [3], [12], [13], [14], to name a few.

Beyesian Network has two basic elements. One is a directed acyclic graphical structure with the nodes of variables and arcs of relations. The other is the conditional probability table associated with each node in the model graph. The table represents the conditional probability distribution of the associated graph nodes. Based on the Bayesian probability inference, the conditional probability can be estimated from the statistical data and propagated along the links of the network structure to the target label. Given a category set $C = \{c_1, ..., c_m\}$ and a set of variable values $\overrightarrow{E} = \{e_1, e_2, \ldots, e_n\}$, the final probability can be calculated as follows and regarded as the indicator of the final decision.

$$P(c_j|\overrightarrow{E}) = \frac{P(\overrightarrow{E}|c_j) \times P(c_j)}{\sum_{i=1}^{m} P(\overrightarrow{E}|c_i) \times P(c_i)} \quad (j = 1, 2, ..., m) \quad (2)$$

According to the basic statistical theory, the joint probability of $\overrightarrow{E}$ can be calculated by the production of local distributions with its parent nodes, *i.e.*,

$$P(\overrightarrow{E}) = \prod_{i=1}^{n} P(e_i|ParentOf(e_i)) \quad (3)$$

If e_i has no parent nodes, $P(e_i|ParentOf(e_i))$ is equal to $P(e_i)$. In our approach, the structure rendering and probability table generation from training sets can be automated through WEKA. Particularly, we use the built-in SimulatedAnnealing algorithm, and the SimpleEstimator algorithm as the threshold control [28].

C. Data grafting

In Stage 1, the input is the vectors of pre-processed terms of summary part (with pre-labelled possibility for supervised learning), and the output is the levels of possibility; while the input of Stage 2 is the set of features both of the other fields as well as the extracted feature from the previous stage.

There can be a format mismatch between the output of the first stage and the input of the second stage. We propose a technique, called data grafting, to smooth the linkage of the two stages. The linkage of the data is not so straightforward as its first glance. During the text classification stage, the report ID is not included into the classifier, yet the only input is the instances with lists of stemmed terms, as well as the pre-labelled levels (from supervised learning). Therefore, we need to track the partition strategy of instances for training and prediction in WEKA and merge the results with other features from the corresponding reports. We assume the partition is based on the n-fold strategy which is commonly used in practice. The grafting technique is given in Algorithm 1.

The input to the algorithm is the instance set with full features where summaries have been preprocessed (with a predefined label of 'possibility' for training), and the output is the instance set updated with the predicted 'possibility', i.e., the required format of the second stage.

After n-fold partitioning based on a given strategy (Line 3), we incorporate WEKA's classification APIs to initialize the classifier (Line 4). During the iteration of each partition, we extract the required features from the original instances (Line 8-9) and set the target classification index (Line 10-11). Then we train the classifier based on the partition (Line 12). After that, each instance in the partition is predicted and the result values are stored and used to update the test partition (Line 13-16). Finally, the updated test partition is merged with the returned instance set (Line 17). Thus the identification of the corresponding instances is realized through the array index values of the input instance set and the extracted instance set.

Data: $instSet$:full-featured instance set, and n:integer
Result: $instSetWPred$:merged with prediction result
begin
  $instSetWPred \longleftarrow \emptyset$;
  split $instSet$ into n $parts$ based on given strategy;
  /* Initialize the classifier with WEKA's build-in Constructor */
  $classifier \longleftarrow$ new $NaiveBayesMultinomialText()$;
  foreach $part[i] \in parts$ **do**
    $testSet \longleftarrow part[i]$;
    $trainset \longleftarrow instSet - part[i]$;
    /* extracting the summary and possibility features */
    $extTestSet \longleftarrow extractSumPoss(TestSet)$;
    $extTrain \longleftarrow extractSumPoss(TrainSet)$;
    /* set feature index 1, i.e., 'possibility' as the classification target label */
    $extTestSet.setClassIndex(1)$;
    $extTrainSet.setClassIndex(1)$;
    $classifier.buildClassfier(extTrainSet)$;
    foreach *inst[j]* $\in$ *extTestSet* **do**
      $inst[j].possibility \longleftarrow classifier.classifyInstance(inst[j])$;
      update possibility value of j^{th} instance $\in testSet$ with $inst[j].possibility$;
    end
    $instSetWPred \longleftarrow instSetWPred \cup testSet$;
  end
end

Algorithm 1: Data grafting algorithm

IV. Experiments

To evaluate our approach, we collected more than 3,200 samples randomly from five large open source projects. Across all the projects, these samples are labelled manually. Firstly, the summary parts are analyzed alone and their likelihood to be a corrective bug report are rated by the individual inspectors. Then other fields are taken into consideration and the final decision is made; a majority-vote is taken to resolve the possible conflicts. These samples are divided into training set and testing set based on the 10-fold cross validation method. In the following, we describe the experimental settings, the performance metrics, and the evaluation results in sequence.

A. Setup

TABLE I. Bug reports info in our experiments

Projects	No.(bug:nonbug)	BTS	Up-to-date
Mozilla(core)	539(343:196)	Bugzilla	3/31/2014
Eclipse	693(473:220)	Bugzilla	3/31/2014
JBoss	573(365:208)	Redhat Bugzilla	4/1/2014
Firefox	620(412:208)	Bugzilla	3/20/2014
OpenFOAM	795(535:260)	Mantis	2/24/2014

Table I summarizes the bug report information used in our experiments. Since the original data set of the baseline work [7] was unavailable to us and still to be consistent, we used the same repositories, i.e., Mozilla[4], Eclipse[5] and JBoss[6]. In addition, the number of our samples are at a similar quantity level (actually, we deliberately collect a bit more to make the classification more stable and thus the comparison more reliable). The other two cases, i.e., Firefox and OpenFOAM, are also included to demonstrate the broader feasibility of our approach. Four out of the five projects use Bugzilla (or its variants); the other one uses Mantis[7]. Across all the five cases, we only consider those reports in *RESOLVED* or *CLOSED* status to avoid duplicates, and preclude those bug reports that are automatically generated and submitted, which is the same strategy used in [7].

Our analysis considers multiple fields of a given bug report, i.e., *textual summary, severity, priority, component, assignee, reporter*. As aforementioned, the textual summary part has been processed in the first stage, and the extracted features are combined with the rest and fed into the classifier in the second stage. The parsing, extraction and combination of the dataset are done automatically through programs. We use Multinomial Naive Bayes text classifier in Stage 1 and Bayesian Network classifier in Stage 2. The WEKA version employed in our experiments is 3.7.11.

B. Metrics

In [7], Antoniol et al. use the evaluation criteria from information retrieval metrics. As we need a way of measuring the performance of our approach and compare with previous models, we are interested in the *precision* and *recall* rates. Using these two metrics as evaluation criteria is a common procedure in related work and the areas of both information retrieval and statistics.

Precision is used to measure the exactness of the prediction set, while recall evaluates the completeness. Precision and recall can be expressed mathematically:

$$precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (4)$$

$$recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (5)$$

Based on the precision and recall, we can calculate the F-measure as below, which denotes the balance and discrepancy between precision and recall.

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} \quad (6)$$

To give an overall performance evaluation, we use the weighted average value of *F-measure* of both categories by the proportions of instances there are in that category. Equation 7 describes the formula to derive the average *F-measure*, where the average F-measure is denoted as $f\text{-}m_{avg}$, F-measure of bug (nonbug) as $f\text{-}m_{bug}$ ($f\text{-}m_{nonbug}$), and number of bug (nonbug) as n_{bug}(n_{nonbug}).

$$f\text{-}m_{avg} = \frac{n_{bug} \times f\text{-}m_{bug} + n_{nonbug} \times f\text{-}m_{nonbug}}{n_{bug} + n_{nonbug}} \quad (7)$$

C. Performance

Based on the above settings, we conduct a suite of experiments—consisting of three parts—to validate of our

[4]http://www.mozilla.org
[5]https://www.eclipse.org
[6]https://www.jboss.org
[7]http://www.mantisbt.org/

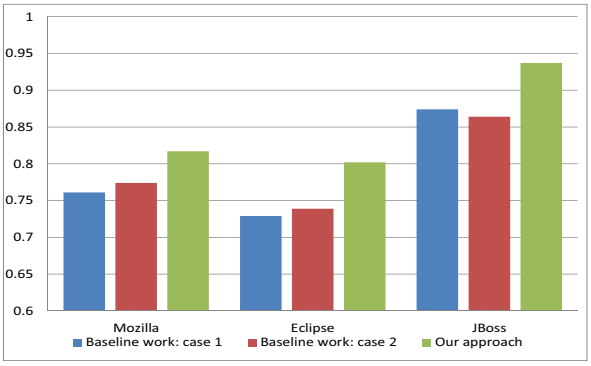

Fig. 3. Comparison with the baseline work (case 1: top 20/20/50 features for Mozilla/Eclipse/JBoss; case 2: top 50/50/100 features for Mozilla/Eclipse/JBoss)

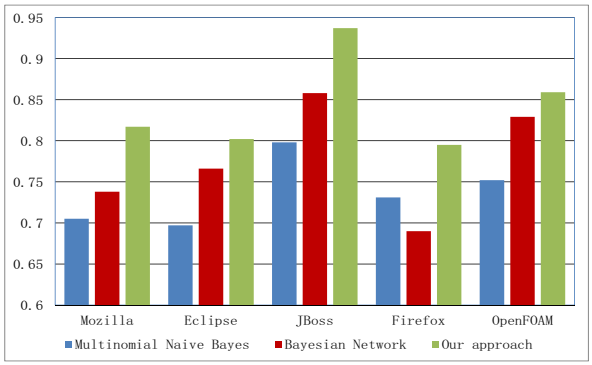

Fig. 4. Comparison in Experiment 2

approach. The first experiment is to compare with the baseline work [7] and conducted on the reports from the same three projects; the second one is to compare with the performance of individual classifiers in each stage with their corresponding features; and the third one is also to compare with the performance of a single classifier but with all the relevant features. The last two experiments are conducted on the reports from all the five projects.

1) Experiment 1: We follow the same experimental settings described in Antoniol et al.'s paper. To make the comparison as objective as possible, we use the same classification algorithm (Naive Bayes, ADTree, Logistic regression), the same preprocessing technique (tokenization and stemming, but no stop-word removal), the same attribute selection strategy (symmetric uncertainty) and parameters (20-50-100 features), and the same validation method (10-fold cross validation).

Table II compares the experimental results based on the approach presented in [7] with that of our approach. There are two cases for the feature selection in [7]. In the first case, the top 20 features recommended by the symmetric uncertainty method are selected for Mozilla and Eclipse, and 50 features for JBoss; while in the second case, the top 50 features are selected for Mozilla and Eclipse, and 100 features for JBoss. Since more features are included in the second case, the results are generally better with a slight exception for JBoss. In the baseline work, the reported best results are derived from the logistic regression model. We faithfully repeated the experiments. For space limitations, the values of the other two classification models, i.e., ADTree and Naive Bayes, are not included.

From the table we can see that our approach consistently outperforms the baseline work across all the three projects in terms of the weighted average F-measure, namely 4.3% in Mozilla, 6.3% in Eclipse, and 6.3% in JBoss. Figure 3 illustrates the comparison. Although the value of the absolute increase is not impressive, if we take the high base value into consideration, the enhancement is still satisfying, since there is limited allowable improvement space left in the case study. Particularly in the case of JBoss, we improve the weighted average f-measure of *bug* from 0.898 to 0.952 and of *nonbug* from 0.831 to 0.910. These results indicate that our approach is highly applicable in practice.

2) Experiment 2: Our approach essentially is a hybrid of prediction models (or classifiers), with each suitable for input features of different nature. We need to answer the question of whether or not the combination of the underlying models performs better than the individual ones. To evaluate the feasibility as well as the generalizability, we also include another two large open source projects in our experiments, i.e., Firefox and OpenFOAM. From now on in the paper, we consider all the five cases. Since we combine Multinomial Naive Bayes model and the Bayesian Net model, we need to evaluate the performance of these two underlying models with the corresponding input features. Concretely speaking, for Multinomial Naive Bayes, the input is the solely textual part of the report; for Bayesian Net, the input is the rest features without the 'possibility' level extracted from the textual part.

The performance results of the component models are given in Part (b) of Table III; while the results of our approach are given in Part(a). From the table, we can see, as to these individual models, that neither one consistently performs better than the other in the sense of weighted average of f-measure across these five cases. Concretely, we have increased the performance from 73.8% to 81.7% for Mozilla, from 76.6% to 80.2% for Eclipse, from 85.8% to 93.7% for JBoss, from 73.1% to 79.5% for Firefox, and from 82.9% to 85.9% for OpenFOAM. Moreover, either for *bug* or for *nonbug* per se, our model outperforms the best of them in any of the cases. The comparison of weighted average f-measure is described by Figure 4.

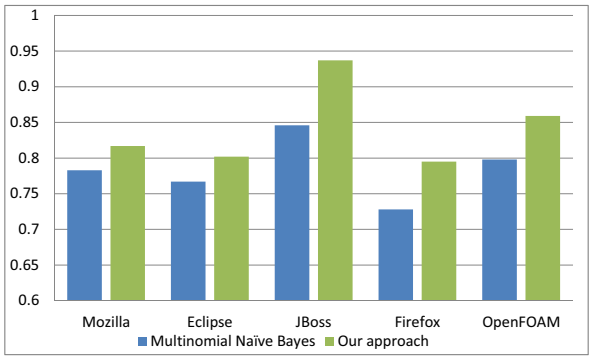

Fig. 5. Comparison in Experiment 3

(a) Prediction results by our approach

	Mozilla			Eclipse			JBoss		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
Bug	0.818	0.930	0.870	0.828	0.907	0.866	0.934	0.970	0.952
Nonbug	0.839	0.638	0.725	0.749	0.595	0.663	0.943	0.880	0.910
Average	0.826	0.824	0.817	0.803	0.808	0.802	0.937	0.937	0.937

(b) Prediction results by Antoniol et al's approach

		Mozilla			Eclipse			JBoss		
Features		Precision	Recall	F-Measure	Precision	Recall	F-Measure	Precision	Recall	F-Measure
case 1	Bug	0.753	0.988	0.855	0.761	0.941	0.841	0.920	0.877	0.898
	Nonbug	0.955	0.434	0.596	0.741	0.364	0.488	0.800	0.865	0.831
	Average	0.827	0.787	0.761	0.754	0.758	0.729	0.876	0.873	0.874
case 2	Bug	0.765	0.977	0.858	0.771	0.924	0.840	0.900	0.885	0.892
	Nonbug	0.921	0.474	0.626	0.714	0.409	0.520	0.804	0.827	0.815
	Average	0.822	0.794	0.774	0.753	0.760	0.739	0.865	0.864	0.864

TABLE II. EXPERIMENT 1: COMPARISON WITH THE BASELINE WORK USING THEIR BEST PREDICTION MODEL

(a) Prediction results with extended cases by our approach

	Mozilla			Eclipse			JBoss			Firefox			OpenFOAM		
	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M
B	0.818	0.930	0.870	0.828	0.907	0.866	0.934	0.970	0.952	0.809	0.925	0.863	0.860	0.953	0.904
N	0.839	0.638	0.725	0.749	0.595	0.663	0.943	0.880	0.910	0.792	0.567	0.661	0.876	0.681	0.766
A	0.826	0.824	0.817	0.803	0.808	0.802	0.937	0.937	0.937	0.803	0.805	0.795	0.865	0.864	0.859

(b) Prediction results using single models with the corresponding partial features in the 2^{nd} experiment

		Mozilla			Eclipse			JBoss			Firefox			OpenFOAM		
Type		Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M
MNB	B	0.735	0.880	0.801	0.750	0.877	0.809	0.821	0.879	0.849	0.769	0.879	0.820	0.773	0.936	0.847
	N	0.680	0.444	0.537	0.586	0.373	0.456	0.758	0.663	0.708	0.664	0.476	0.555	0.769	0.435	0.555
	A	0.715	0.722	0.705	0.698	0.717	0.697	0.798	0.801	0.798	0.734	0.744	0.731	0.772	0.772	0.752
BN	B	0.779	0.831	0.804	0.797	0.905	0.848	0.861	0.932	0.895	0.751	0.818	0.783	0.843	0.925	0.882
	N	0.665	0.587	0.623	0.712	0.505	0.590	0.860	0.736	0.793	0.561	0.462	0.507	0.808	0.646	0.718
	A	0.737	0.742	0.738	0.770	0.778	0.766	0.860	0.860	0.858	0.687	0.698	0.690	0.832	0.834	0.829

(c) Prediction results using a single model (MNB) with all the relevant features in the 3^{rd} experiment

	Mozilla			Eclipse			JBoss			Firefox			OpenFOAM		
	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M
B	0.787	0.927	0.851	0.792	0.924	0.853	0.871	0.890	0.881	0.771	0.864	0.815	0.818	0.916	0.864
N	0.815	0.561	0.665	0.745	0.477	0.582	0.800	0.769	0.784	0.646	0.490	0.557	0.770	0.581	0.662
A	0.797	0.794	0.783	0.777	0.782	0.767	0.845	0.846	0.846	0.729	0.739	0.728	0.802	0.806	0.798

TABLE III. EXPERIMENT 2 AND 3: COMPARISON WITH INDIVIDUAL PREDICTION MODELS (B:BUG, N:NONBUG, A:AVERAGE, BN:BAYESIAN NETWORK, MNB:MULTINOMIAL NAIVE BAYES)

3) Experiment 3: In the previous experiment, we split the features based on their structuredness and feed them into corresponding models. The separation leads to an incomplete information source to the individual models and thus possibly results in the degradation, making the performance incomparable with ours. Therefore it is reasonable to compare to other models but with complete information. In Experiment 3, we mix all the relevant features, both structured and unstructured parts of the bug reports and regard them as holistic text input to a single classifier.

To be consistent with the textual classification stage of our approach, we also use Multinomial Naive Bayes model as the comparative target classifier with the same parameters as set in our approach in the experiment. The result is given in Part (c) of Table III. From the table, we can see that our approach still performs better. Concretely, we have increased the weighted average f-measure from 78.3% to 81.7% for Mozilla, from 76.7% to 80.2% for Eclipse, from 84.6% to 93.7% for JBoss, from 72.8% to 79.5% for Firefox, and from 79.8% to 85.9% for OpenFOAM. Figure 5 illustrates the comparison results. Again, either for *bug* or for *nonbug* per se, our approach also outperforms the single model with holistic text input in terms of f-measure.

In addition, combined with Part (b) of Table II, we can observe that the inclusion of all features into a solely textual Multinomial Bayes classifier outperforms that of only-description parts in the cases of Mozilla and Eclipse, but this does not hold in the case of JBoss. Likewise, comparing Part (c) with Part (b) of Table III, the classifier with all features only has the advantage over those with incomplete features

in the cases of Mozilla and Eclipse. But the advantage does not exist in the rest three cases. This leads to the hypothesis that the inclusion of all features does not necessarily generate better prediction results against a sub-set of certain features. The reason is perhaps that not all features are created equal and it suggests that some less important features can be discarded until optimal classification performance is reached. This '*less is more*' phenomenon somehow coincides with the findings in [30] and indicates interesting research questions, but a detailed investigation is out the scope of this paper.

V. Discussion

Although the reports are under the name of "bug", the term is actually misused widely. Recent empirical studies already reveal a notable amount of bug reports are actually non-relevant with bugs [8], [9]. As a minor contribution of our paper, we complement and confirm their findings with more cases. All the five cases are not included in their work. Indeed, the reports are randomly selected and non-bug reports account for around 34 percentage in total. Therefore, the implicit assumption that these bug reports are all about software defects is not true. Should this assumption be invalidated, the effectiveness of the proposed approaches relying on it might be violated. To mitigate the problem, we present a novel approach by combining classifiers to automatically identifying whether a given report is bug-relevant or not. The bug reports generally have two parts which have different nature. Our approach allows for a composition of prediction models of with each model tuned for the characteristics of underlying data. A suite of comparison experiments demonstrate the feasibility and effectiveness of our approach[8]. Despite the fact that in the first stage, the classification of textual parts brings additional cost compared with single models, we alleviate the problem by utilizing summary part only to reduce the effort and the data grafting algorithm to link with the second stage automatically.

In our approach, we follow the general text pre-processing steps, i.e., tokenization, stop-word removal and stemming. However, in the baseline work, Antoniol et al. explicitly mention that stop-word removal step is omitted. Thus in the first experiment, we prepare the input accordingly. But we also test their models' performance with the removal of common stop-words. To our surprise, the results are nearly the same. After examination, we find the reason is that they use *symmetric uncertainty* attribute selection method. The inclusion of stop-words does not affect much the top 20 or 50 feature list.

In the evaluation part, we use the weighted average f-measure across the three experiments as a comprehensive indicator for the performance, taking f-measures of both buggy reports and non-buggy ones into consideration based on their proportion. Since computing the average can be sometimes misleading. For example, class one has 100 instances and we achieve a f-measure of 30%, and class two has 1 instance and we achieve f-measure of 100%. When taking the average, we get 65%, and actually inflate the f-measure. But the weighted average is around 30.7%, which is much more realistic measure of the performance of the classifier. In our cases, since non-bug reports have less portion compared with bug reports, they contribute less to the overall average value.

[8]The experimental data have been uploaded and can be retrieved at: http://moon.nju.edu.cn/twiki/pub/ICSatNJU/YuZhou/data.zip

Threats to Validity. In this part, we discuss some potential threats to the validity of our experiment based on the guidelines proposed in [31].

A. Construct Validity: This aspect reflects to what extent the operational measures represent what is investigated according to the research questions. In our paper, we try to answer the question of whether or not a given bug report is a corrective one through a hybrid approach combining both text mining and data mining techniques. We solely rely on the information sources from typical bug reports—consisting of both structured and unstructured textual fields, without resorting to other information, for example, change logs, source code snippets, etc. We combine two classifiers according to the different characteristics of the underlying fields, and train them to make predictions. Although the original datasets of the baseline work are unavailable, we choose the same projects and faithfully repeat their experiments. On the one hand, we use the comprehensive weighted average F-measure to evaluate our approach and make comparisons; on the other hand, we also compare the F-measure values of both buggy and non-buggy reports.

B. Internal Validity: This aspect of validity concerns whether the causal relation are properly demonstrated. In our approach, the included fields of bug reports are frequently reported relevant in the literature, for example, textual parts, severity fields, priority levels, etc. [26], [3], [11], [12], [13]. Another concern of this aspect is the potential bias introduced in the data set. As studied in [32], selection bias is a prevalent problem in software engineering research. Although we follow the general heuristics [8] in the field to manually classify them, and use simple majority-vote to resolve the conflicts, it is inevitable that the manual classification introduces partial subjectivity in our approach. However, we attempted to alleviate this problem by applying an independent inspection strategy by each examiner and the majority-vote mechanism to resolve conflicts.

C. External Validity: This aspect of validity relates to the generalizability issues of the results on the datasets other than those studied in the experiments. We deliberately include two more cases with two different kinds of BTS besides the three essential cases studied in the baseline work. As far as these five cases are concerned, our approach indeed outperforms other individual classifiers as illustrated in our experiment suite. But like other empirical studies, we cannot guarantee that such advantage still holds in other projects. Particularly, all the five cases studied belong to the open source software category, and open source data can differ quite a lot from proprietary software data. Thus we do not know whether other proprietary bug reports share similar characteristics. Nevertheless, we can say that our hybrid model is actually a framework, basically open and not bound to any particular classifiers. Despite the fact that we leveraged Multinomial Naive Bayes Network classifier and Bayesian Network classifier for explanation purpose, new models with better performance for each stage can be integrated into our approach. Another issue is about the number of bug reports collected from projects. More instances generally promise more stable performance [3]. As argued in the baseline work, a sample size of 600 issues was sufficient to ensure a confidence level of 95% percent and a confidence interval of 10% for precision and recall in the context of the

study [7]. Accordingly, we controlled the sample size at the similar quantity level but deliberately included more in our paper.

D. Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. During our manual preparation of experiments, we need to collect a *golden set* for study. Since submitters of bug reports are of various quality, the examination process highly depends on the expertise of reviewers. We omitted the computer-generated bug reports as well as those with inconsistent fields to mitigate the problem. Our tool support is mainly from WEKA—a widely used software suite for data mining. The algorithm of data grafting has been tested against the manually merging results.

VI. Related Work

Antoniol et al. [7] first address the misclassification problem of bug reports and demonstrate the feasibility of classifying textual descriptions to automatically differentiate between them. Pingclasai et al. later propose an alternative approach to classifying the bug reports based on topic modeling [33]. Both consider the textual parts only. Kim et al [9] recognize the pervasive existence of noise in current data collection process and study the effect of these noises on the prediction models. Herzig et al. [8] manually examine more than 7000 issue reports, give a fine-grained classification and analyze the misclassification implications of these reports on earlier studies, but they do not provide the automated classification support. Textual classification techniques have also been employed to classify software microblogs [34], tag recommendation [35].

Apart from the work on binary classification of bug reports, a number of research have been conducted to predict other properties. For example, Lamkanfi et al. [3] propose a text mining based approach to automatically predict the severity of a bug report. Similarly, Menzies et al. [11] present a method named SEVERIS to assist developers in assigning severity levels to bug reports. Both of them solely consider the textual descriptions of the bug reports. But their work demonstrate the correlations between these parts. Moreover, textual descriptions are also leveraged to detect duplicate bug reports [36], [37], to facilitate bug triage [26], [38], to predict the time [1], [39] or the effort to fix a bug [40]. But most of the aforementioned work only consider the information in the textual part of bug reports.

Some work also combine other sources apart from bug reports to facilitate the prediction task. For example, in [41], Wang et al. leverages the execution information to find the most similar bugs. In [42], Matter et al. use the similarity between a given bug report and source code changes to facilitate bug triage. The main difference of such kind from ours is that we only consider the information available in bug reports.

Only a few work study the implication on the inclusion of multiple factors in bug reports or multiple dimensions in bug repositories. Their results indicate a positive gain in terms of performance. Examples include [12], [43], [16]. In [12], Shihab et al. combine the dimensions of work habits, bug reports, bug fix and team. Based on these dimensions they create decision trees to predict whether a bug will be re-opened or not. In [43], Sun et al. fully utilize the information available in a bug report including not only the textual content, but also the non-textual fields, such as product, component, version, etc. to detect duplicate bug reports. In [16], Tian et al. extract multiple factors, such as temporal, textual, author, related-report, severity, and product, and propose a new classification model to predict the priority levels of a given bug reports. Although the underlying idea is similar with us, we differ in both the problem target and the approaches. As far as we know, very few prior work, if any at all, have ever proposed the combination of models to classify bug reports.

VII. Conclusion and Future Work

In this paper, we presented a hybrid approach of combining text mining and data mining techniques of bug report data to identify corrective bug reports. This way we reduce the noise of misclassification (i.e., filtering bug reports that are not corrective) and support better performance of bug prediction. Our work is essentially a multi-stage classification approach—a particular kind of ensemble learning techniques [44]—composed by a set of specific learning algorithms with the aim of outperforming the constituent individual ones. This also brings broad opportunities to apply other kinds of similar techniques into the classification problem in future. We validated our approach with an empirical study of five open source projects, compared it with the baseline work, and with the underlying individual classifiers in three experiments.

Although more case studies and a larger corpus of samples would certainly help to substantiate whether our approach consistently outperforms the underlying individual classification algorithms, our approach is not dependent on any particular algorithm. It is inherently open and those new algorithms can thus be integrated into our framework. For example, better textual classifiers can substitute Multinomial Naive Bayes classifier used in our approach. To the best of our knowledge, this is the first work on the combination of classifiers in bug report prediction. In future work, we plan to investigate how many training examples are needed in each stage to give stable prediction results and theoretically analyze to what extent the combination of separate models can enhance the overall performance given the performance of the individual ones, as well as the noise resistance ability of our approach.

Acknowledgment

The authors would like to thank Prof. Yun Niu, Prof. Dechuan Zhan, Dr. Xinye Cai and anonymous reviewers for the improvement on the paper. This work was partially supported by the NSF of China under grants No.61202002, No.61170043, No.61272083, and the Collaborative Innovation Center of Novel Software Technology and Industrialization in Jiangsu Province.

References

[1] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[2] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

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

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340–355, 2005.

[5] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 31–41.

[6] M. DAmbros, H. Gall, M. Lanza, and M. Pinzger, *Analysing software repositories to understand software evolution*. Springer, 2008.

[7] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "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*. ACM, 2008, p. 23.

[8] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[9] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 481–490.

[10] P. S. Kochhar, T.-D. B. Le, and D. Lo, "It's not a bug, it's a feature: does misclassification affect bug localization?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 296–299.

[11] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[12] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 249–258.

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

[14] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 25–35.

[15] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 27–30.

[16] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 200–209.

[17] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[18] Y. Yang, "An evaluation of statistical approaches to text categorization," *Information retrieval*, vol. 1, no. 1-2, pp. 69–90, 1999.

[19] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[20] A. Hotho, A. Nürnberger, and G. Paaß, "A brief survey of text mining." in *Ldv Forum*, vol. 20, no. 1, 2005, pp. 19–62.

[21] J. Kittler, M. Hatef, R. P. Duin, and J. Matas, "On combining classifiers," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 3, pp. 226–239, 1998.

[22] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 127–134.

[23] D. D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *Machine learning: ECML-98*. Springer, 1998, pp. 4–15.

[24] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 337–345.

[25] K. Chaturvedi and V. Singh, "Determining bug severity using machine learning techniques," in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*. IEEE, 2012, pp. 1–6.

[26] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[27] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1.

[28] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[29] Y. Zhou, M. Wuersch, E. Giger, H. Gall, and J. Lu, "A bayesian network based approach for change coupling prediction," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 27–36.

[30] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *Software Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 552–569, 2013.

[31] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[32] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 411–414.

[33] N. Pingclasai, H. Hata, and K.-i. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*. IEEE, 2013, pp. 13–18.

[34] P. K. Prasetyo, D. Lo, P. Achananuparp, Y. Tian, and E.-P. Lim, "Automatic classification of software related microblogs," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 596–599.

[35] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[36] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 499–510.

[37] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[38] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[39] P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 523–526.

[40] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 1.

[41] 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 *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 461–470.

[42] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 131–140.

[43] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[44] C. M. Bishop *et al.*, *Pattern recognition and machine learning*. springer New York, 2006, vol. 1.