Expert Systems With Applications 167 (2021) 113808

Contents lists available at ScienceDirect

Expert Systems With Applications

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

# SPBC: A self-paced learning model for bug classification from historical repositories of open-source software

Hufsa Mohsin, Chongyang Shi *

*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China*

ARTICLE INFO

*Keywords:*
Bug triaging
Defect localization
Self-paced learning
Bug report analysis
Bug classification

ABSTRACT

One of the areas most in need of improvement in the field of automated bug fixing, localization and triaging systems is that of an effective categorization, as this would bugs to reduce the time, cost and effort required to locate, assign and fix the bug. The existing approaches depend upon the textual similarity of the bug description and category in a given reported bug; accordingly, the challenges of unstructured bugs, technical terms, versatile ways of reporting the same bug, the diverse nature and sizes of datasets etc. are often overlooked. Consequently, this limits the classifier performance to a specific type of dataset, resulting in classification inefficiency. To this end, we propose a novel **S**elf-**P**aced **B**ug **C**lassifier (SPBC) that is capable of locating the target categories from the bug description of the historical data, maintained by multiple open-source software packages (Bugzilla, Mentis, Redmine). The proposed model introduces a self-paced back-traceable algorithm, controlled by a self-paced regularizer, which classifies textually independent bug descriptions with weighted data-independent tokens (the easy samples). Later on, the regularizer sets comparatively hard samples for textually dependent classification by capturing intra-class and inter-class discrimination features from bug descriptions, based on the weighted similarities of words; this is done with the help of a Key Feature Identification Matrix (KFIM), a Non-Independent and Identically Distributed (NIID) matrix. Easy-to-hard self-pace learning, integrated with textually dependent and independent classification, makes SPBC capable of simultaneously enhancing the effectiveness and robustness of intelligent systems through a substantial increase in precision (5–15% on average). The main advantage of SPBC is that it targets the spatial relationship between the data and the system, which makes it an apt learner of data and allows it to maintains sample insertion into the classifier at a controlled pace. Additionally, it maintains stability, which is not affected by the dataset's dimensionality and traits. As is evidenced by the experimental results on four different datasets from open-source projects, our model outperforms the baseline and state-of-the-art methods through a single-stroke solution with improved accuracy and stable performance (average 95% precision and 4% decrease in kappa); hence, it is significant for improving intelligent bug fixing and triaging systems.

## 1. Introduction

With the advent of modern technology, the ultimate goal is to build intelligent systems for important tasks one encounters in real-life scenarios. At the same time, the aim is to build this software to be error-free; this requires a system capable of automatically diagnosing and fixing the bugs during a piece of software's entire life cycle. Research on historical repositories serves as a foundation for the study and understanding of the bugs and their nature, Thung, Shaowei, et al. (2012). One such important repository is bug reports maintained by bug tracking applications, which log issues, bugs and defects and help the software industry, researchers and practitioners in predicting failure-prone projects, making managerial decisions in the software industry and in source code change prediction applications (Hattori et al., 2009; D'Ambros and Lanza, 2012), also assisting developers in testing, recommending bugs that should be reopened (Zimmermann et al., 2012), facilitating the co-evolution of test and production codes (Zaidman et al., 2008), bug report summarization (Li, Jiang, et al., 2018) and much more. Bug report classification serves as basis of many intelligent systems and their applications, such as automated fault localizers (Zhang et al., 2017), bug triagers (Anvik and Murphy, 2011), intelligent bug fix systems (Razzaq et al., 2018; Kim and Whitehead, 2006),

* Corresponding author.
*E-mail addresses:* hufsa.bit@yahoo.com (H. Mohsin), cy_shi@bit.edu.cn (C. Shi).

https://doi.org/10.1016/j.eswa.2020.113808
Received 18 June 2019; Received in revised form 28 July 2020; Accepted 29 July 2020
Available online 19 August 2020

automatic defect categorization (Thung, Lo, et al., 2012) and debug recommenders (Yu et al., 2008).

We emphasize that understanding the bug type represents the first and most time-consuming step in the process of bug triage, localization and even bug fixing. Consequently, effective bug categorization is the foundation of addressing the defects and loopholes in a particular system in mi-nimal time and with the least resource wastage. Moreover, few approaches have been proposed solely for the purpose of bug type identification, which do not benefit effective and stable performance due to the challenging issues involved in bug reporting. Moreover, the performance of classifiers differs from one dataset to another. Hence, we present the need for designing a stable model that is less biased and capable of improving the performance of classifiers.

In the literature, bug report categorization is part of a major research effort that deals with attribute similarity between pieces of information in a bug report, as a basic assumption. For instance, Almhana et al. (2016) presented a bug localization approach based on the lexical similarity of bug descriptions and API documentation, while (Zhang and Zhong, 2016) generated bug report summaries using a deep belief network. The work by Wang et al. (2016) automatically learns semantic features from a source code for defect prediction, while (Zhang et al., 2016) used K-Nearest Neighbors (KNN), based on instance similarity between a new bug report and similar reports from the historical repositories; the main aim of their research is to assign a bug report to the relevant developer. Zhang et al. (2017) used a word co-occurrence approach that depends on textual similarity to identify buggy files from a source code. Very few works have addressed this crucial problem as a separate task. Limsettho et al. (2016) proposed a bug categorization framework based on textual similarity without the need for labeled data. Neelofar et al. (2012) presented a naïve Bayes classifier for bug report classification.

Other existing text classification approaches also depend on lexical similarity. For example, Guzman-Cabrera et al. (2009) presented a semi-automated lexical approach for the non-English, language independent classification of a web corpus. Some approaches are word vector- and co-occurence-based; (Amensisa et al., 2018), for instance, used a bi-gram vector space model for large text document classification, while (Zhang and Zhong, 2016) used word vector representation for short text classification. Zheng and Wang (2018), however, proposed a text mining approach that operates by combining Convolutional Neural Network (CNN) and Support Vector Machine (SVM) with active self-paced learning. Their classifier captures classes by progressively annotating unlabeled samples and verifies error-labeled sample data.

However, bug categorization is hindered by a number of factors, including the different nature of data repositories, multiple reporting methods, the unstructured nature of natural language noise in bug reports (e.g., stop words, links, stack trace, technical information), false alarms, feature set handling and various other issues (Shivaji, 2013). Since every dataset contains different feature sets, with some providing many similar word tokens for inferring the bug type, while others totally lack similarity among bug categories and descriptions. Therefore, text similarity-based approaches can make the classifier dependent and the results produced on multiple types and various nature data can lead to incorrect categorization, instability in results and so on. Moreover, the size of bug descriptions or summaries in a bug report is small and contains a very small number of words from which feature set can be formed. After the preprocess the number of word tokens is reduced even further, resulting in a conceptually disjoint set of tokens. Hence, concept-similarity- or co-occurrence- based approaches may not benefit from complete efficiency improvement. Furthermore, Neural Network (NN)-based approaches are subject to tuning for each type of data and requires large datasets for training; therefore, the chances are high that an entirely different performance will be obtained on multi-faceted datasets.

Because bug categorization is the basis of every bug recommendation system, there is a vital need to perform this task exclusively and effectively. Current successful applications of **S**elf-**P**aced **L**earning (SPL) for classification and clustering problems (Lu et al., 2015; Ma et al., 2017; Zhao et al., 2016) serve as motivation for building a bug report classifier that takes into account both the attribute–value similarity of a structurally disjoint feature set and textual interdependence integrated with SPL in order to solve the issues with the current approaches.

In the light of the above, we propose a novel classifier, SPBC, by exploiting the Non-Independent and Identically Distributed (NIID) (Cao, 2016) approach with SPL, for data of multiple natures and sizes. The method involves employing two-staged processing in order to deal with textually similar and non-similar bugs separately. To achieve this objective, we introduce a back-traceable algorithm that facilitates bug categorization, independent of textual similarity. Furthermore, we employ **K**ey **F**eature **I**dentification **M**atrix (**KFIM**) to form a feature matrix that stores one key discriminative feature for each bug. The key features are then used to classify all textually dependent bugs. Finally, we propose a novel self-paced model, that operates systematically, in stages, to classify textually dependent and independent bugs while maintaining performance and stability. The model evaluates the connection between the data input order and classifier performance by inducing easy and hard samples in a controlled and systematic manner.

The main impact of the model lies in the fact that it is capable of enhancing the performance of systems that are part of intelligent bug fixers or recommenders without being affected by the nature of the datasets. As evidenced by the experimental results, the Kappa coefficient and the accuracy of other existing methods decreased significantly (12% to 35%), while for SPBC, only a 4% decrease was observed on 10-fold cross-validation. Hence, the **B**ack **T**raceability (BT) algorithm and KFIM are conducive to improved and stable accuracy. Furthermore, KFIM helps in capturing multiple bug categories. The major contributions of this research can be summarized as follows:;

- The model presents a novel, robust and effective classifier that is capable of enhancing the performance, of the bug fixing and triage systems. It does this by learning systematically, at its own pace, to capture bug categories from traits that are bug reports with varying traits that are obtained from open-source software.
- The model introduces a key feature identification matrix and a novel, back-traceable self-paced learning algorithm to classify textually independent and dependent bug descriptions in controlled stages, thereby enabling it to solve the problem of performance inconsistency that impacts existing approaches.
- The proposed model has deliberately been implemented on four multi-faceted and multi-dimensional datasets (sizes ranging from approximately 2800 to 1 million). Our evaluation results show that the proposed approach is accurate (94% to 99% precision on average, with an AUC up to 0.85) and stable (with an AUC up to 0.85 and Kappa of 0.9). The model also presents a stable response (only a 4% decrease in accuracy on various datasets) when compared to the existing approaches (12–35% decrease).

### 1.1. Motivation

In the context of our research, we believe that there is a lack of research that tries to provide the automatic labeling of bug types. Current approaches aim to support the bug triage process without considering categories and performs differently for various data-sets, being textually dependent. Hence, there is a need for devising a different mechanism altogether, for identifying bug types. Consequently, inspired by the benefits SPL has offered in the fields of computer vision and graphics, we incorporated the self-paced induction of data into our two-stage classifier to obtain the target bug categories by keeping the robustness, stability and effectiveness intact.

However, the way in which SPL is applied in our model is totally different from the applications currently being used in computer vision

research, while the problem under consideration is also entirely different, hence, to the best of our knowledge, the way in which we incorporate the SPL for bug categorization makes our method a novel one. The self-paced regularizer in our approach controls the data sample induction, and the nature of the classification also depends on this parameter. Furthermore, the combination of textually dependent and independent classification integrated with SPL adds to the novelty of the model. Below is an example that quickly explains its usage.

Bug reports, which are maintained by bug tracking software, contain multiple attributes of information. Data contained in a generic bug report is shown in Fig. 1. However, different software programs maintain information in different ways. Examples of bug reports, maintained by Bugzilla, Redmine and Mentis, are presented in Tables 1, 2 and 3 respectively. The Redmine and Mentis bug reports contain a separate column ("category") to represent the type of each bug, while the Eclipse project repository tags the type of bug in square brackets with the description in the column "shortshortdesc" and does not have a separate category or type column; for instance, the [SFS] bugs shown in Table 1. Moreover, the text size of each reported bug is not the same, and the category assignment keywords are also different from the words in description. Some bug categories have greater textual similarity for words with a description/summary, whereas others have a generic token set with little or no such similarity. In the latter case, the chances are that defining keywords may be found in multiple classes, which leads the existing textual similarity-based approaches to misclassification or performance variations in varied datasets. Furthermore, the size of the token set (the number of words in each description) is not particularly large, and after obtaining a reduced feature set, approaches that learn semantically might also not perform well on multi-faceted data, as the semantics are disturbed to a considerable extent (Fig. 2). Hence, keeping the above factors in mind, a dataset-independent framework is required to achieve stability and improvement in the classifier's operations.

To achieve our objective, the obtained reduced feature set, after removing the noisy data (links, stack trace, punctuation, stop word removal, etc.), is separated into textually similar and non-similar categories by the self-paced learner. For example, the *UI* category in Table 3 is one type of sample that has no similar words in the *subject* and category name *UI*. All these stages are controlled by the self-paced regularizer parameter $\mu$. In the first phase of SPL, the model extracts those samples that include the tokens with the highest weight, instead of matching any text in the "description" and "category", and classifies these samples separately -independent of the textual similarity- with the help of the proposed back-traceable self-paced algorithm. The other phase is executed by taking out the token samples with the lowest weight and classifying them – again, independent of the text. For rest of the bug samples, the proposed model first obtains the defining feature – the feature that best corresponds to that bug type – by using the KFIM, which exploits the NIID. Finally, this feature is mapped to the bug category with the highest weight for this keyword. In this way, both

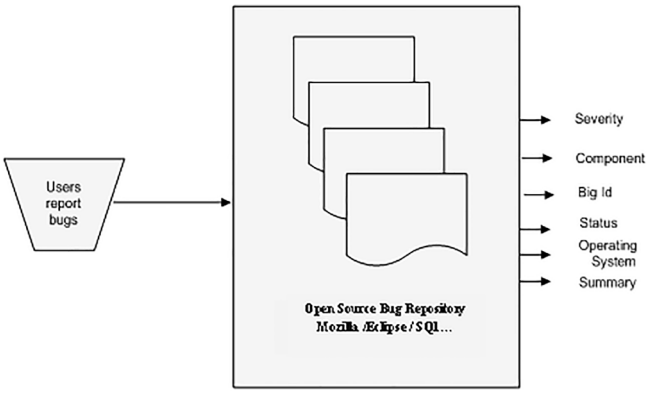


**Fig. 1.** General information stored in a bug report of open-source software.

textually similar and non-similar types are classified creating a stable mechanism that is not affected by the nature of the dataset or keyword type. As noted above, many approaches rely either on textual similarity or semantics, but rarely both. Furthermore, a huge amount of text is available for feature extraction for classes with a common English vocabulary, whereas bug report description are very short texts containing technical terms. Hence, our approach takes all these factors into account by incorporating SPL, our model becomes more effective than to others. The detailed operation of the model is explained in Section 3.

## 2. Related works

Research into historical bug reports or bug triaging has been attempted in a number of ways over the past several years in an attempt to solve multiple problems. Approaches used to solve the problem under consideration have ranged from simple statistical methods to complex NN and even more sophisticated deep learning models. Some of the research that relates to our work, in one form or other, is discussed in the following sub-sections.

### 2.1. Bug prediction with text mining approaches

Extensive surveys on bug prediction techniques, issues and challenges were presented by Thung, Shaowei, et al. (2012), Aggarwal and Zhai (2012), Allahyari et al. (2017), Zhang et al. (2016), Jie et al. (2015), Qiu et al. (2017). These surveys have highlighted many issues in bug tracking systems and bug reports, ranging from the duplication of bugs to their severity and priority; however, none of them has comprehensively discussed bug labeling techniques based on summaries or short descriptions in machine learning systems. For the task of categorizing bugs from historical bug report data, some widely used machine learning algorithms include SVM, NB, Decision Tree (DT), and NNs (Wang et al., 2016; Allahyari et al., 2017; Elmishali et al., 2018). LDA topic modeling (Limsettho et al., 2016; Zhou et al., 2016; Kalyanasundaram and Murphy, 2012), has also been used to determine feature similarity, but it is not exactly applicable to the problem under our consideration. Deep Belief Network (DBN) (Jiang et al., 2018), Abstract Syntax Trees (ASTs) (Wang et al., 2016; Almhana et al., 2016), Deep Neural Networks (DNNs) and rVSM are applied for defect prediction from the source code or to automatically learn semantic features from the token vectors extracted from programs. Xiaobing Sun (2019) also proposed an IR-based bug localization technique that operates by extracting bug patterns from version-related bug reports. Moreover, Chen et al. (2019) incorporated NNs to present a novel concept of bug entities and their relationships from bug reports.

Zhang et al. (2017) performed a study to detect bug misclassification by combining both text mining and data mining techniques to bug reporting data in order to automate the prediction process. The aspect of their work that most relates to ours is the part that leverages text-mining techniques to analyze the summary of bug reports and classifies them into three levels of probability: bugs, non-bugs and average bugs. The authors have used Bayesian networks and multinomial NB. Some research discussed here was based on historical data but solved different problems than ours: bug prioritization issues, bug fix time, etc. On the other hand, some research that focused on the defect classification used run-time repositories; for instance, code segments that are of an entirely different nature.

### 2.2. Self-paced learning approaches

This section highlights the applications of SPL in the computing field. Guzman-Cabrera et al. (2009), presented a semi-supervised self-paced model derived from a web corpus, which is used for language-independent text categorization. The work of Lu et al. (2014) is also based on the self-learning of auxiliary data from the internet and incorporates transfer learning. Ma et al. (2017) suggested a semi-

**Table 1**
Eclipse bug report maintained by Bugzilla and GitHub. Bugs are tagged with corresponding types in the bug description instead of a separate column, for instance, [sfs].

| Bug_id | Product | Component | Priority | Bug_severity | Bug_status | Op_sys | Short_short_desc |
|---|---|---|---|---|---|---|---|
| 266750 | E4 | Resources | P3 | normal | RESOLVED | All | [sfs] fix javadoc for options parameter |
| 267704 | E4 | Resources | P3 | Normal | RESOLVED | Windows XP | [sfs] Externalize names of SFS extension points to allow translation |
| 276349 | E4 | Resources | P3 | major | RESOLVED | Linux | [sfs] Exception when running SfsExamplesTestSuite |

**Table 2**
Redmine bug report with a separate "category" column for each bug

| Id | Category | Severity | Status | Summary | |
|---|---|---|---|---|---|
| 24975 | Signup | 3 | Minor | Resolved (atrol) | Due to some reason not sign up |
| 22315 | Markdown | 6 | Minor | Assigned (joel) | Markdown converts " to " within code blocks and inline code |
| 25006 | ui | 1 | Minor | Feedback | Changing order of View Boxes makes them dissapear |

**Table 3**
Mentis bug report with separate "category" column for each bug and a "subject" rather than "summary" of bugs.

| # | Subject | Category |
|---|---|---|
| 1100 | Custom Fields should have a Flag "Show in compressed Issueheader" | Custom fields |
| 1086 | Fine grained permissions | Permissions |
| 1081 | Add Activity block to My Page | UI |

[sfs] fix javadoc option parameter
[sfs] Eliminate issue discover FindBugs
[sfs] External name SFS extension point allow translation
[sfs] Extension point schema file added binary build
[sfs] Exception run SfsExampleTestSuite

**Fig. 2.** Reduced token set of Eclipse bugs after applying the preprocessing steps (stopword removal, stack trace removal, stemming, etc.).

supervised approach, which trains a classifier on two different views. The limitation of removing the assumptions is dealt with by the authors. Pe et al. (2016) present an effective and robust learning method that is capable of capturing inter-class discriminative patterns, maintaining reliability and providing a fully corrective optimization for classification. Real-world datasets are used for the experiments. Li et al. (2018) used the SPL framework for multi-label learning on real-world data. Xu et al. (2015) discussed a SPL approach applied to clustering and dealt with the bad local minima problem. These authors applied a smoothed weighting scheme to define the easy and hard samples. Real-world experimental datasets were used for experimentation. Fan et al. (2017) suggested a neural data filter, utilizing a DNN rather than heuristic-based SPL. Zheng and Wang (2018) introduced a self-paced CNN algorithm for text classification applied to news data. The algorithm first initializes the classifier using a few annotated samples and extracted text features using CNN, and it ranks the unlabeled samples according to their importance.

While all these works use SPL models in one way or another, none has solved the bug classification problem in bug tracking systems, which is the issue under consideration in this work. Furthermore, these works are tested on web corpora, news data and other similar datasets containing large amounts of text with a large number of potential features that can help in classification. In our case, however, the text that requires classification contains comparatively few words and technical terms, which makes the situation entirely different. Moreover, deep learning approaches are data-set-specific, i.e., trained specifically for every set of data and parameters, meaning that an approach tuned for one might not be equally impressive for another. Hence, our approach differs from the above-mentioned models in terms of the nature of the experimentation as well as the methodological approach. Furthermore, identifying easy and hard samples for the self-paced regularizer is also achieved differently in our model compared to the progressive acquisition of data samples in the approaches discussed above. Our regularizer controls the data sample injection and determines the type of classification (i.e., similarity-dependent or independent classification).

## 3. The proposed model

An overview of the proposed model is presented in Fig. 3 and the symbols used are presented in Table 4. The following subsection provides a detailed discussion of the model.

### 3.1. Problem formulation and model framework

Let $X=Y^n$ be an n-dimensional feature space and $C = c_i,\ldots,c_k$ be a set of possible classes. Consider a training set $D = (b_i, c_i)$, $i^n_{=1}$, where $b_i = b_{1i}$, $\ldots,b_{in} \in X$ is the i-th feature and $c_i = c_{i1},\ldots,c_{ik} \in C$ is the label vector associated with $b_i$. The goal of the bug classifier is to learn $c_i$ from $D$ to predict unknown bugs. Most bug classifiers attempt to map a bug $b_i$ to a class $c_i$ based on a certain classifier function $\delta$, as stated below in Eq. (1).

$$\forall\delta({}_1{}^n b_i) \leftarrow C_k \tag{1}$$

Here, $\delta$ is normally calculated as the weight of tokens needed to identify the textual similarity of words in the class and the inter-class difference. For the proposed classifier function $\delta$ of our model, let us define $T = t_i,\ldots,t_k$ as the set of tokens for each $c_i$; moreover, let $W = w_{1i}$, $\ldots,w_{nk}$, be the weight and $F = f_i,\ldots,f_k$ be the frequency vectors for every $t_i$. Let $z = \delta(x_i);l_i$ be the key feature identification function that yields the key features $l_i \in l_1,\ldots,l_n$, generated from the window of words in each class $ci$, depending on the function $\delta$, which is defined as follows:

$$\delta_i = max_{0\leqslant x\leqslant n} \sum_{i=0}^{k-1} w^t_{i+1} log \frac{n}{f^t_{i+1}} \tag{2}$$

The mapping function Ω, maps each feature to the corresponding bug type as follows:

$$\Omega_i = max_{0<k<n}\, w_{(t)}.l_i(c_i) \tag{3}$$

The model operates in a self-paced fashion such that it first attempts to map each $b^m_i$, having a similarity index $s \equiv 0$ or $\lessapprox$ threshold, i.e., the bugs with the highest or lowest weight in each class $c_i$. Subsequently, it classifies the remaining $b^n_m$, having a similarity index $s \gtrapprox$ threshold to class $c_i$. The general optimization problem of SPL (Li, Wei, et al., 2018) is given by the equation below:

$$\sum_{i=1}^{n} v_i L(w, x_i, y_i) + f(V;\gamma); \tag{4}$$

Here, $x_i, y_i \in D$ and $L(w, x_i, y_i)$ are the loss function, $v_i$ is used for calculating importance, and $f(V;\gamma)$ denotes the self-paced regularizer function. For the problem under consideration, we have defined our own functions according to our situation with relevant parameters. The

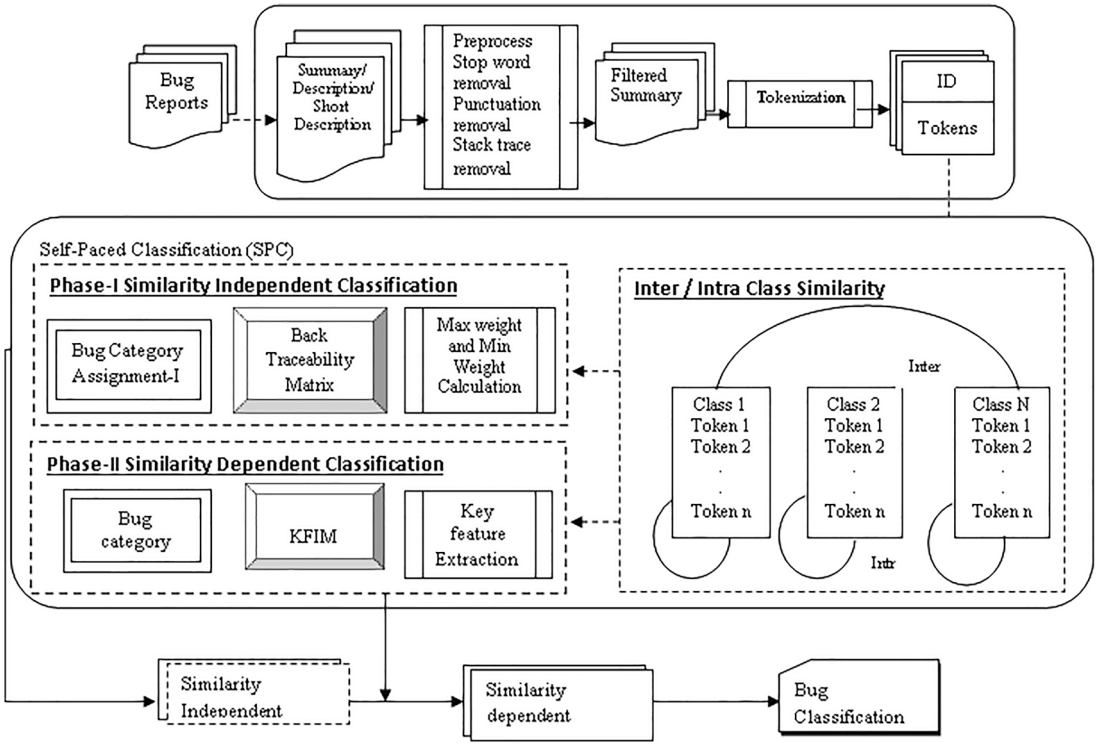


**Fig. 3.** Proposed model of SPBC.

**Table 4**
Symbol table.

| Symbol | Description |
|---|---|
| $X$ | n-dimensional feature space |
| $C$ | Set of possible classes |
| $D$ | Training set |
| $bi$ | ith feature |
| $ci$ | Label vector |
| $\delta$ | Classifier function |
| $T$ | Set of tokens |
| $W$ | Weight vector |
| $z$ | Key feature identification function |
| $l_i$ | Key feature |
| $\Omega$ | Feature Mapping Function |
| $\mu$ | Regularizer function |

regularizer function will operate in three steps controlled by the parameter $\mu$, as given by Eq. 5:

$$\mu\forall({}_1{}^m b_i) + \mu\forall({}_{m+1}{}^t)b_i + \delta\forall({}_t{}^n)b_i \leftarrow C_k \quad (5)$$

### 3.2. Proposed Algorithm

.

**Algorithm 1.** SPBC-I: Similarity-independent classification

1: Input $D=(b_i,c_i)$, $i=1..n$, $T=t_1\ldots t_n$,$W=w_{1i},\ldots w_{nk}$ and $F=f_i,\ldots,f_n$
2: Output $BT(d_i, t_i,f_i,w_i)$, $C_k$, class labels of $m+t$ bugs where $m < t < n$
3: ($BT$) Preprocess($D$)
4: **for** $b_i$ in $D$ **do**
5: Remove stopwords, stacktrace, punctuation from $b_i$
6: Apply stemming on $b_i$
7: **end for**
8: **for** filtered $b_i$
9: **while** $D \neq \varnothing$ and $D$ has classes $C_i$
10: Form training classes $C$ contacting each $b_i \in C$
11: **end while**

(*continued on next column*)

(*continued*)

**Algorithm 1.** SPBC-I: Similarity-independent classification

12: **end for**
13: **for** $c_i$ in $C$
14: Split each filtered $b_i=b_{i1},\ldots,b_{in} \in X$ to $T_i$, space delimited set of $T_i$
15: Calculate $\sum_{c=1}^{l} f_c=\frac{f_{t1}+f_{t2}+\ldots+f_{tm}}{t_n}$
16: Calculate $W_i=\log[(nt_c)\ (\frac{f_{t1}+f_{t2}+\ldots+f_{tm}}{t_n})]$
17: Form $BT$ with $f_i$ and $w_i$ for each $t_i$ having bug Id $d_i$
18: **end for**{//Matrix to map bugs independent of Text}
19: Return $BT(d_i,t_i,f_i,w_i)$
20: ($l_i$) KeyFeatureIdentification ($T$,$W$,$f$)
21: **for** $b_n$ in $D$
22: get each token $t_i$ in $b_n$
23: calculate $\delta_i=\sum_{j=0}^{k-1} w_{j+1}^t \log\frac{n}{f_{j+1}^t}$
24: compare $t_i,t_{i+1}\ldots t_n$
25: **if** $t_i \leftarrow \max\ 0 < k < n\ \delta_i$
26: $l_i=t_i$;
27: **else**
28: **if** $w_i < \max w_k$
29: check $t_{i+1}$ {// repeat for all tokens}
30: **end if**
31: **end if**
32: **end for**
33: Return $l_i$ as feature set {//key fetaure set for KFIM}
34: SPC($d_i$, $t_i$,$f_i$,$w_i$) {//The key feature set obtained by the function ($l_i$) KeyFeatureIdentification ($T$,$W$,$f$) function above is further utilized by Algorithm 2 for classifying the similarity-dependent bugs. For textually independent mapping, function SPC ($d_i$, $t_i$,$f_i$,$w_i$) makes use of $BT$ to map the bugs based on highest weights, regardless of any comparison in classes.}
35: **for** $\mu = 1$, $\mu\forall({}_1{}^m b_i)$
36: get $\max_{0<m<k} \sum_i^m w_i\ M(t_i,l_i,c_i)$
37: **if** $d_i$ in $D$ matches $d_i \in BT$
38: label each $b_m$ as $c_k$ w.r.t. $BT(\mu;d_i)$
39: **else**
40: repeat till $c_i = \varnothing$
41: **end if**
42: $C_k \leftarrow \forall({}_1{}^m)b_i)$, where $m < t < n\ \forall\ D$
43: **end for**

(*continued on next page*)

(*continued*)

**Algorithm 1.** SPBC-I: Similarity-independent classification

44: **for** $\mu\forall(({}_{m+1}{}^{t})b_i)$, $\mu = 2$
45: get $\min_m < t < k\sum_{i=m+1}^{t} f_i M(t_i, l_i, c_i)$
46: **if** $d_i$ in $D$ matches $d_i \in BT$
47: label each $b_m$ as $c_k$ w.r.t. $BT(\mu;di)$
48: **else**
49: repeat
50: **end if**
51: $C_k \leftarrow \mu\forall(({}_{m+1}{}^{t})b_i)$, where $m < t < n\forall D$
52: **end for**
53: Return $C_k = \mu\forall(({}_{1}{}^{m})b_i) + \mu\forall(({}_{m+1}{}^{t})b_i)$ {//The SPC function obtains category mapping for the bugs, independent of description text, by locating the maximum weight feature and tracing it back with the help of IDs stored in BT to map the category against each record. Remaining bugs are mapped by Algorithm 2 via comparisons against each class. BT again prevents the overriding of the correct mapping obtained by Algorithm 1}

Every natural language problem must undergo the common steps of data cleaning, stop word removal and stemming in order to obtain a reduced dataset that is fit for further processing. In our case, the bugs are cleaned of the punctuation, links, special characters, etc. as a first sub-step of pre-processing. Subsequently, stop words are also removed; since these words do not support any learning process and are not meaningful, they should be removed before further processing. Finally, stemming is applied to reduce each word to its stem. This reduces the feature set considerably, and it helps in finding similarity weights. These steps are outlined in Algorithm 1. Hence, for every bug $b_i$, in the dataset $D = (b_i, c_i)$, the pre-processing function $(BT)Preprocess(D)$ is applied as shown in Algorithm 1 (Line #1 to Line #7). This yields a filtered dataset $D_f$ as shown in Fig. 2 above. Prepossessing.java code class serves the purpose. $D_f$, undergoes more steps, as more explained in the sections below.

### 3.2.1. Training class formulation

After preprocessing, we obtain a filtered summary $D_f$ for each bug $b_i$. For the training model, we need the bugs from every class $C_k$. Each bug $b_i \in C = c_i, \ldots, c_k$ of similar classes are collected to form the training classes $C = c_i, \ldots, c_k$. Table 5 presents a sample class of bug type "compatibility". The steps are explained in the preprocessing function of Algorithm 1 (Line # 8 to Line # 12). Furthermore, we need to tokenize each $b_i \in b_{i1}, \ldots, b_{in}$ into a space-delimited set of tokens $T = t_1 \ldots t_k$ so as to form a vector space matrix for frequency and weight calculations (Algorithm 1: Line # 13 to Line #16). Training.java code class is used for creating class tables and maintain the relevant information.

### 3.2.2. Proposed key feature identification

In order to extract the defining features $l_i \in l_1, \ldots l_n$, for each bug $b_i$ out of all tokens $t_i \in t_1 \ldots t_k$, the bug contains, a vector space matrix is formed that is similar to the one used for NIID (Cao, 2016), where frequency $f$ is the intra-attribute coupling of tokens (i.e., the occurrence frequency within one class). The weight $w_i$ of each token yields the inter-attribute coupling and is calculated by the independence and its importance to that class, as shown in Eq. 6. The matrix so formed (KFIM) contains Y-dimensional classes $c_1 \ldots c_n$ and X-dimensional tokens $t_{i1} \ldots t_{nm}$, with weight attribute $w_{i1} \ldots w_{nm}$, and with values between 0 to 1, as obtained by the Equation below:

$$W_i = \| \sum_{j=0}^{k-1} w_{j+1}^{t} log(\frac{n}{f_{(j+1)}^{t}}) \| \tag{6}$$

which gives the normalized weight between 0 and 1.

The function $(l_i)KeyFeatureIdentification(T, W, f)$ in Algorithm 1 (Line #20), extracts the features; here, $T, W$ and $F$ are the tokens, weights and frequencies stored by $BT$ obtained through the function $(BT)Preprocess(D)$, respectively. In the key feature identification phase, every $t_i$ of each $b_i$ is first evaluated for intra-class importance (Algorithm 1: Line # 21 to Line # 24). Finally, based upon the maximum weight $max\sum_{j=0}^{k-1} w_{j+1}^{t} log(\frac{n}{f_{(j+1)}^{t}})$ and using Ω, the mapping function defined in Eq. 3, the token $t_k$ is extracted as the defining feature $l_i$ of each $b_i$ returned by the function *KeyFeatureIdentification* in Algorithm 1 (Line # 25 to Line # 33). BackTraceabilityDes.java and FeatureExtraction.java code classes are used for this purpose. Detailed steps are presented in Algorithm 1, while Table 5 depicts the features extracted for a few bug types.

### 3.2.3. Proposed self-paced classification

Before proceeding to the details, the preliminaries will first be explained. For the self-paced classifier to operate, a back-traceability vector matrix $BT = (t, w, f, d_i)$, is built; this matrix can trace a feature $l_i$ back to its origin in $D$ using the identity $d_i$, as explained by the function $SPC(d_i, t_i, f_i, w_i)$ in Algorithm 1 (Line # 34), which operates on the $BT$ attributes in phases controlled by $\mu$. For $\mu = 1$, both weights and frequency are calculated for each distinct word in the whole vector space and class vector (Line # 35 to Line # 42). Let $f_i$ be the frequency of the i-th token of $b_i$ in class $c_i$. Frequency and weight calculations are given by Eq. (7) and Eq. (8).

$$\sum_{c=1}^{k} fc = \frac{(f_{t1} + f_{t2} \ldots + f_{tm})}{t_n} \tag{7}$$

Here, $f_{t1} + f_{t2} \ldots + f_{tm} = f_t \in c_k$ refers to the number of similar tokens in each class; moreover,

$$W_i = log[(n_{tc} \frac{(f_{t1} + f_{t2} \ldots + f_{tm})}{t_n})] \tag{8}$$

where $n_{tc}$ is the similar token count in all classes $c_k$.

Table 6 provides the gist of this matrix formulation. The tokens $t_i$ for each $b_i$ are stored with their corresponding bug ID numbers $d_i$, weight $W_i$ and frequency $f_i$, each time they appear in the class. This helps the SPL phase to locate a particular bug during the mapping phase, as shown in Algorithm 1, and the output is displayed in Table 6. The first phase $\mu\forall({}_{1}{}^{m}b_i)$ of the SPC (Self-Paced Classifier) is defined in Eq. (9).

$$max_{0<m<k} \sum_{i=1}^{m} w_i M(t_i, l_i, c_i) + BT(\mu; d_i) \leftarrow Ck \tag{9}$$

Here, we have used a mapping function $M(t_i, l_i, c_i)$ rather than $L$ in Eq. 4. This mapping function with the help of $BT(\mu; d_i)$ maps each extracted feature $l_i$ and the tokens $t_i$ in each class $c_i$, corresponding to the maximum weight $w_i$, as this is the easiest sample to map correctly.

For $\mu = 2$, the second phase $\mu\forall({}_{m}{}^{t}b_i)$ of SPC maps the bugs based on the mapping function $M(t_i, l_i, c_i)$ and $BT(\mu; d_i)$, where the frequency of token $t$ in class $c_i$ is minimum for the defining feature $l_i$ (Algorithm 1: Line # 44 to Line # 52). Eq. 10, explains the working for $\mu = 2$. This phase also maps the bugs that are not yet classified, as depicted by $m < t < k$ in Eq. 10.

$$min_{m \leqslant t \leqslant k} \sum_{i=m+1}^{t} f_i M(t_i, l_i, c_i) + BT(\mu; d_i) \leftarrow Ck \tag{10}$$

**Table 5**
The defining feature of every class extracted from the filtered summary obtained by feature extraction.

| Feature | Summary |
|---|---|
| NPE | NPE java outline |
| NPE | NPE thrown multiple console |
| view | outline view toolbar close reopen |
| java | part creat java implement |
| menu | New context menu outline |
| list | ctrl + F6 honour activat list |
| list | ctrl + F7 honour activat list |
| NPE | NPE try launch Product Run Configurat Dialog |
| CSS | CSS platform |

**Table 6**
Back-Traceability Training Matrix containing frequency and weights for each class, along with the bug Ids to facilitate the process of locating a bug.

| Bid | Token | Frequency | Weight |
|---|---|---|---|
| 147 | Open | 4 | 4 |
| 147 | Files | 2 | 5 |
| 147 | Drag | 1 | 6 |
| 150 | URISyntaxExcept | 1 | 6 |
| 150 | Workspace | 4 | 4 |
| 150 | loc | 2 | 5 |
| 151 | Space | 1 | 6 |
| 151 | NPE | 11 | 3 |
| 151 | Code | 1 | 6 |
| 152 | Assist | 1 | 6 |

For $\mu = 3$, the classifier function $\delta\forall({}_t{}^n b_i)$, as explained in Eq. 2, is processed; this will map the rest of the bugs for $t < k < n$.

#### 3.2.4. Similarity independent classification

The self-paced learner works in steps controlled by the parameter $\mu$, as explained above. For $\mu\forall(({}_1{}^m)b_i), \mu = 1$, i.e., the first phase of self-paced learning, the model considers the word with the highest weight in each class computed by the mapping function $\sum_{i=1}^{m} w_i M(t_i, l_i, c_i)$. This makes use of *BT*, the bug traceability matrix, to get the tokens $t_i$, their corresponding weights $w_i$ and features $l_i$. $max_{0<m<k}\sum_{i=1}^{m} w_i M(t_i, l_i, c_i)$ maps $b_i$ in $c_i$ to the corresponding class $C_k \leftarrow \delta\forall(({}_t{}^n)b_i)$, where $m < t < n \forall D$. Similar to SP1, the next phase of SPL is repeated by mapping the factor with the lowest weight. For $\mu\forall(({}_{m+1}{}^t)b_i), \mu = 2$; again, the bug type is assigned to the corresponding class via back-tracing by computing $min_{m<t<k}\sum_{i=m+1}^{t} f_i M(t_i, l_i, c_i)$. The mapping function maps $C_k \leftarrow \mu\forall(({}_{m+1}{}^t)b_i)$, where $m < t < n$, for all $D$. Hence, $C_k \leftarrow \mu\forall(({}_1{}^m)b_i) + \mu\forall(({}_{m+1}{}^t)b_i)$ yields the class labels for the bug types in all classes containing the highest and lowest weight $w_i$. The complete steps are outlined in Algorithm 1.

#### 3.2.5. Similarity-dependent classification

.

**Algorithm 2.** SPBC-II: Similarity-dependent classification

1: Input $BT(d_i,t_i,f_i,w_i)$, $D$, $l_i$
2: Output $C_k$, class labels of $b_i$ where $m < t < n$ {//Uses back-traceable matrix and key features in KFIM for each bug to map remaining bugs}
3: get $d_i$, $t_i$,$f_i$,$w_i$ from $BT$ for each $C_k$ in $D$.
4: **for** $\delta\forall(({}_t{}^n)b_i)$, $\mu = 3$
5: compute $\Omega_i(\max_{0<m<k} w_t.l_i(ci))$
6: **if** $\Omega_i = \max_{0<m<k} w_t.l_i(c_i)$ AND $C_k$=null $\in D$ OR $C_k \neq b_t$
7: label each $b_{m+1}$ as $c_k$
8: **else**
9: repeat $\forall\ c_i)$
10: **end if**
11: $C_k = \delta\ \forall_t^n\ b_i$, where $m \lessdot t \lessdot n$ *forall* $D$
12: **end for**
13: Return $b_i = \mu\forall(({}_1{}^m)b_i) + \mu\forall(({}_{m+1}{}^t)b_i) + \delta\forall(({}_t{}^n)b_i), \forall b_i \in D$ {Returns all bugs categorized}

By SPC($d_i, t_i, f_i, w_i$), the SPC function in Algorithm 1, nearly 10% to 15% of bugs on average are assigned to their corresponding classes. For $\delta\forall(({}_t{}^n)b_i), \mu = 3$ (Line # 4, Algorithm 2), the remaining ones and the bugs that are incorrectly assigned by computing the classification function $\Omega_i(w_t.l_i(c_i))$ are explained in Section 3.1. If the weight $w_i$ of the extracted feature $l_i$ from the feature matrix, is maximum and $C_k$ = null or wrongly classified by SPC (Line # 6 & 7, Algorithm 2), then the bug is assigned a category $C_k \leftarrow \delta\forall(({}_t{}^n)b_i)$, where $m < t < n \forall D$ (line # 11, Algorithm 2) by using the KFIM, explained in Section 3.2.2. The inter-class coupling of features, as given by Eq. 6, is computed. The ith feature $l_i$ is searched against each class $C$, and the class $c_i$ with $max_{0<k<n} w_t.l_i(c_i))$ weight for $l_i$ is mapped to that feature. If two classes have the same weight for token $t_i$ in the KFIM, as explained in Section 3.2.2, then multiple labels are assigned to that bug. This also solves the problem of multi-label classification. Algorithm 2 explains the steps of the process. Weights and IDs are put into a hashmap to simplify and speed up the search. This process of class label assignment continues until every bug in the dataset is assigned to a class. The bugs whose features were not found in any KFIM are tagged as *Not Mapped* and then used in the evaluation phase to compute the accuracy and recall. SplHFBugs.java code class is used to handle the SPL phases and to map each bug accordingly.

## 4. Experiments and results

This section first presents the research questions constructed to evaluate our model, followed by the collection and preprocessing of datasets. Subsequently, the evaluation metrics are presented. Finally, a detailed discussion on the findings obtained through the series of experiments is included. Eclipse Java (Oxygen) is used as IDE with java as the programming language. We formed a package named "spl" for incorporating all the java classes including, Connection.java for connecting the datasets, Prepropressing.java for all the preprocess steps, Training.java for obtaining training class tables, BackTraceabilityDes. java for keeping record of bugs and Ids to be further used in bug back traceability, FeatureExtraction.java for extracting key features from each bug, SplHFBugs.java for controlling SPL phases. EvaluationPRF. java for evaluating the model with accuracy, precision, recall and F-Score metrics.

### 4.1. Research questions

The following are the research questions devised to evaluate the model:

- RQ1: Is SPBC's performance more stable on different datasets than other approaches?
- RQ2: Does the proposed approach outperform other approaches?
- RQ3: Is SPBC's performance robust with different training sizes and different datasets?
- RQ4: Does SPL help to improve the performance of the proposed approach? If so, to what extent?

The first and second research questions (RQ1 & RQ2) investigate the performance and stability of the proposed model on different datasets and in comparison to other approaches. The third research question, RQ3, is formulated to investigate the accuracy variations and robustness of the model with datasets and training samples of various sizes. RQ4, evaluates the effect of SPL on improving the performance of the classifier.

### 4.2. Datasets and preprocessing

The proposed approach has been implemented and evaluated on four different datasets from top bug tracking systems Bugzilla (the data for the Eclipse Project retrieved from https://bugs.eclipse.org/bugs/ while adding keywords to search bugs from 2010 to 2018), Redmine (https://www.redmine. org/issues, containing bugs from 2008 to 2018) and Mentis (retrieved from https://www.mantisbt.org/bugs/view_all_bug_page.php containing bugs from 2004 to November 2018) bug reports and the Git-Hub dataset for the Eclipse project (retrieved from https://github.com/ansymo/msr 2013-bug_dataset, containing 168.000 reported bugs from the Eclipse project until 2013) and 10000 other tagged Eclipse dataset (retrieved from kaggle.com/monika11/bug-triagingbug-assignment/data). The rationale behind choosing these specific software programs, out of many, was that all of them differ to some extent in terms of the way they record the bugs. The Eclipse dataset, derived from both Git-Hub and Bugzilla, has bug classes marked inside the summary, which contains more tokens for each bug than the other datasets. Mentis and Redmine have a separate column for each bug

category. The Redmine dataset has very few classes compared to Eclipse which contain the keywords matching the bug category (i.e. Redmine has very different phrases from the type that is being assigned to the bug). This can affect the accuracy, as the approach is token-based. The characteristics of each dataset are outlined in Table 7. Since bug description is a special form of text with many data types, even for smaller datasets. In addition, it does not have many different words to correspond to a bug type; rather, it has a specific bag of words for each type, containing technical terms or the ones that clearly correspond to that type. Therefore, acquiring huge data records might not bring vast variations in results as the defining features for one class, of our model, do not differ drastically with increased sized of data. Hence, this justifies the selection of smaller datasets.

We collected more than 30,000 tagged bugs from Bugzilla using a keyword search of the major types, we extracted the records that contained tags (i.e., bug labels tagged in square brackets in the "short description" attribute of the dataset automatically). Similarly, over 1 million bugs from GitHub projects were also extracted here, either bugs tagged within the square brackets of "short description" or major bug types were defined in the "component" column of the dataset. Since the format of all the Eclipse datasets was similar hence, we collected the non-duplicate bugs under one heading "Eclipse dataset". Together, these make up sufficient bug reports, with over 1 million labeled records to test the model.

The reason why the tagged bugs were collected from the Eclipse dataset is that these labels help in quickly identifying the nature of a bug without the need to read the text. Hence, they are quite helpful in immediately delegating the bug to a relevant expert for fixing. For huge datasets like Eclipse, we have these tags/labels in the huge number, which correspond to specific types. Out of almost 2000 bug types, we reduced the labels for Eclipse by merging common types; for instance, 886 bug labels are used for one type of training/testing set, indicating the main type, subtype or even sub-subtype of bug. For Redmine and Mentis, moreover, the bugs were already tagged in the available dataset under the "category" containing 52 and 54 bug categories, respectively. Therefore, these datasets were used without further processing to extract the bug labels. These categories/labels/tags are then used to train different classes of bugs, and finally, to compare our results with already tagged bug types in order to calculate accuracy, precision and other measures.

### 4.3. Metrics

In this section, the classification of SPBC is evaluated using a number of evaluation methods. We calculate the accuracy, specificity and robustness of the proposed model on datasets of varying sizes. We also computed the performance of SPBC on different types of datasets. A comparison between our method and baseline methods is also presented in this section. This section defines the metrics used to evaluate the robustness, performance and stability of our model.

#### 4.3.1. Robustness metrics

To evaluate the robustness of the classifier, we used ROC and AUC in terms of the FPR (False Positive Rate) and TPR (True Positive Rate).

**Table 7**
Characteristics of datasets: Three datasets are shown here; since the GitHub dataset also contained Eclipse project bugs, this dataset is combined with Bugzilla under the common name "Eclipse dataset".

| Datasets | Size | Labeled data | Class labels | No. of tokens per record on average | Learning complexity |
|---|---|---|---|---|---|
| Eclipse | 92035 | 32580 | 1486 | 9 | Easy |
| Redmine | 4416 | 4416 | 52 | 4 | Medium |
| Mentis | 2358 | 2358 | 54 | 3 | Hard |

#### 4.3.2. Performance metrics

To evaluate the performance of SPBC, we measured the accuracy and precision of the model in terms of True Predictions (TPs), False Predictions (FPs) and Missed Predictions (MPs).

#### 4.3.3. Stability metrics

To examine the stability of our model, we used recall (Sensitivity), F1 Score and Cohen's Kappa.

These metrics are tested with 10-fold cross-validation and compared with the results obtained on the same testing and training sets to evaluate the stability of the model.

### 4.4. Comparison with other approaches

To evaluate RQ1 and RQ2, we compared our algorithm with the existing approaches. For RQ1, we used kappa statistics to evaluate the stability of SPBC compared to other methods. We used ANOVA for hypthesis testing (RQ2), i.e., to determine whether our approach outperforms others.

Although quite a number of techniques are available in this area, none of them was exactly the same as ours. The approaches chosen for comparison are therefore those that somewhat relate to ours. In the literature, the approaches used specifically for dealing with the problem of bugs in bug reports using summary or description were variants of naïve Bayes (Zhang et al., 2017). NNs have also been recently introduced into the field of bug classification. Although these are mostly used for identifying bugs from source code repositories, however, we have conducted a comparison with Zheng and Wang (2018), in which the authors used CNNs and Active SPL to perform unlabeled text classification; thus, it is related to our model in some aspects. The work of Zhang et al. (2016) is also compared. These authors used KNNs, based on instance similarity between a new bug report and similar reports from the historical repositories. Other approaches chosen, were either widely used bug prediction techniques or text classification techniques (both baseline and state-of-the-art methods).

For comparison, we used the same kind of dataset, similar conditions and environment, where data is trained to produce maximum output. The following sections present the performance analysis by taking samples for testing from the same training set with 10-fold cross-validation to compare the stability on various portions of the dataset.

#### 4.4.1. Performance Comparison with Testing Data Extracted from the Same Training Set

Figs. 4 and 5 present clear evidence that our approach performs better than all existing approaches, when trained and tested on the same datasets. SPBC obtained 99% accuracy and precision on the Redmine and Mentis datasets. Here, it should be noted that the models are tested on known data obtained randomly from the training set; therefore, no considerable variation is observed between the performance of SPBC and other approaches. However, some of the basic approaches exhibit variation on different kinds of datasets, as depicted in Fig. 5. The vertical bars illustrate the performance variation on different datasets for each method. However, Bayesnet, which is used in many bug classification studies, performed poorly on our datasets. It can further be observed that SPBC achieves consistent performance on different datasets, in terms of Cohen's Kappa, which is at 0.96.

To validate the significant difference between the proposed approach and other approaches, we employ (one-way) ANOVA, as all approaches are being applied to the same datasets. This may validate whether the only difference (single factor, i.e., different approaches) leads to performance variations. We conduct the ANOVA in Microsoft Excel using the default settings, and no adjustment is involved. Notably, ANOVA on accuracy, precision, recall, and F- measure is conducted independently, where the unit of analysis is the dataset. Table 8 presents the ANOVA analysis results: here, $F > Fcrit$ and p-value $<$ (alpha $= 0.05$) are true for precision and F-measure. This suggests that the factor (using different

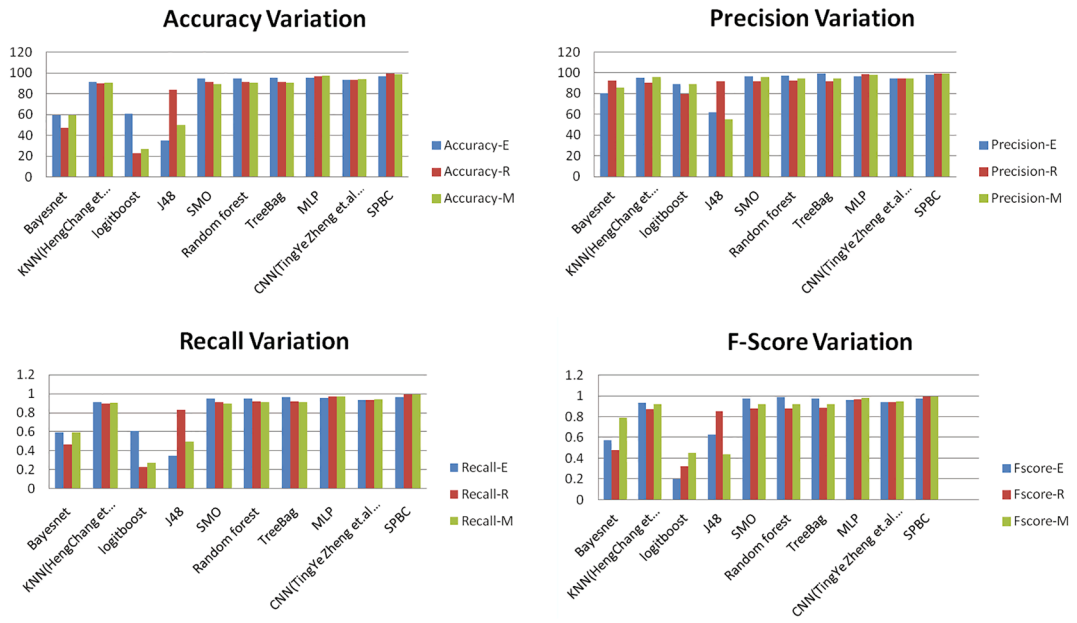


**Fig. 4.** Performance evaluation of the baseline methods with SPBC w.r.t. Accuracy, Precision, Recall and F-measure. Testing and training sets are from the same dataset.

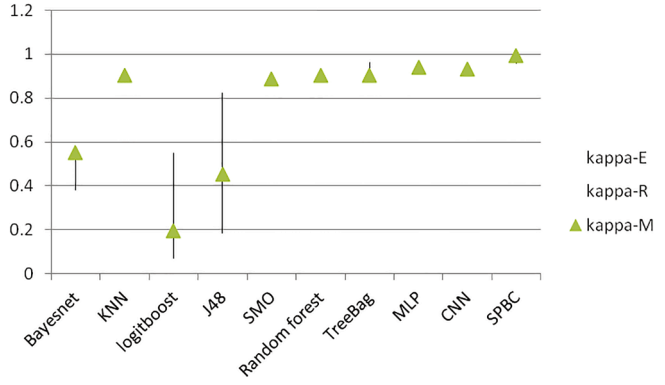


**Fig. 5.** Performance evaluation of baseline methods and SPBC in terms of cohen's kappa. The model is trained and tested on the same dataset. Vertical sticks show performance variation on different datasets. SPBC exhibits a constant performance on different data-sets and is nearly equal to 1.

approaches) exhibits a significant difference in accuracy and F-measure.

#### *4.4.2. Stability comparison with other approaches (on 10-fold cross-validation)*

To compare the stability of approaches on unseen data, we used 10-fold cross-validation. Here, one dataset portion is set aside each time for testing and the remainder is used for training. The difference in accuracy between SPBC and other methods is shown in Fig. 6. SPBC shows a stable performance (only a 4% decrease). However, for cross-validation, the other algorithms performed poorly. The accuracy of all methods exhibits a considerable decrease (ranging from 12% to 35%), with 10-fold cross-validation showing lower stability with variations in data. A comparison of these results and those achieved by SPBC suggests that our model is more stable on different datasets and substantially accurate, particularly on the Eclipse dataset. This is because SPBC works on a constant back-traceable matrix that provides fewer fluctuations with changes in data, as it takes the easy and hard sample for each dataset.

All models chosen for comparison are well known for their ability to solve text mining and classification problems. However, these models performed poorly on our dataset and experimental setup. To investigate

**Table 8**
ANOVA analysis on the comparison.

| Source of Variation | SS | df | MS | F | P-value | F crit |
|---|---|---|---|---|---|---|
| Accuracy | | | | | | |
| Between Groups | 13179.97 | 9 | 1464.442 | 12.73664 | 1.73E-06 | 2.392814 |
| Within Groups | 2299.573 | 20 | 114.9786 | | | |
| Total | 15479.55 | 29 | | | | |
| Precision | | | | | | |
| Between Groups | 2056.646 | 9 | 228.5162 | 4.768642 | 0.001758 | 2.392814 |
| Within Groups | 958.4121 | 20 | 47.92061 | | | |
| Total | 3015.058 | 29 | | | | |
| Recall | | | | | | |
| Between Groups | 1.32114 | 9 | 0.146793 | 12.97969 | 1.48E-06 | 2.392814 |
| Within Groups | 0.226189 | 20 | 0.011309 | | | |
| Total | 1.547329 | 29 | | | | |
| F-Score | | | | | | |
| Between Groups | 1.279843 | 9 | 0.142205 | 15.36806 | 3.7E-07 | 2.392814 |
| Within Groups | 0.185065 | 20 | 0.009253 | | | |
| Total | 1.464909 | 29 | | | | |

Where, SS = sum of squares, DF = degree of freedom, MS = mean square, F = F-test (used for comparing models), and p-value determines the significance

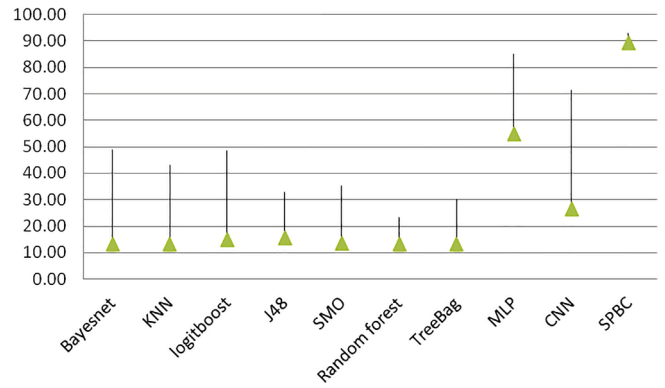


**Fig. 6.** Stability evaluation of baseline methods and SPBC on 10-Fold cross-validation, in terms of accuracy. The accuracy of all methods has exhibited a considerable decrease (ranging from 12% to 35%) with 10-fold cross-validation, illustrates lower stability with variations in data. Vertical sticks show the performance variation on different datasets for each method. Here, the performance of SPBC can be observed as stable (a decrease of only 4%).

further, we examined our data closely and found that the summary contains only a small number of words. Following the removal of stop words and stemming, these numbers are reduced further, and their semantics are disturbed to quite an extent. Hence, the approaches that operated by finding the semantic meanings of a phrase did not perform well. It is preferable to guess the keyword that defines a certain class. Our algorithm calculates the relevance of keywords aided by both supervised learning and the self-learning approach. We analyzed our model even further to assess other performance factors, as described in the following sections.

We validated our model for robustness by evaluating it on different datasets with varying proportions of training data. This was achieved by gradually decreasing the proportion of the training set and evaluating the remainder of the data.

### 4.5. Robustness analysis

#### 4.5.1. ROC and AUC

This section validates RQ3. Fig. 7 represents the results of the ROC analysis with the Eclipse, Redmine and Mentis datasets. Observations started from 90% for the training sets and gradually reduced by 10% each time for all datasets, to observe their behavior. Correct and false cumulative calculations were made to produce TPR and FPR. The AUC is almost the same for all datasets, while the ROC curve exhibits similar behavior. However, Redmine shows a lower AUC compared to the other two datasets. The AUC for Eclipse is 0.85, while that of Mentis is 0.84, which is almost the same for the two datasets. Fig. 7 graphically represents the ROC curves for multiple datasets. It is evident from the results that the model fits Eclipse dataset the best. For the Redmine dataset, this cumulative value of AUC drops to 0.74. We investigated the reason by closely examining the dataset. Our findings suggested that the Redmine dataset was a particularly difficult dataset for the model to learn as it contained quiet varyingly stated bugs in each class, which makes the model training difficult.

The curve for Redmine is lower than the other two datasets. However, the point to note here is that the training dataset ratio is as low as 10%, which will definitely affect the curves. Moreover, the dataset variability is high in the case of Redmine; still, the AUC is 0.74, as suggested by Fig. 7, which is above 0.5, which means that the model is still capable of differentiating between classes more than 70%. Hence, to conclude, our model has performed fairly well on multiple nature datasets and is scalable to increased numbers of testing sets.

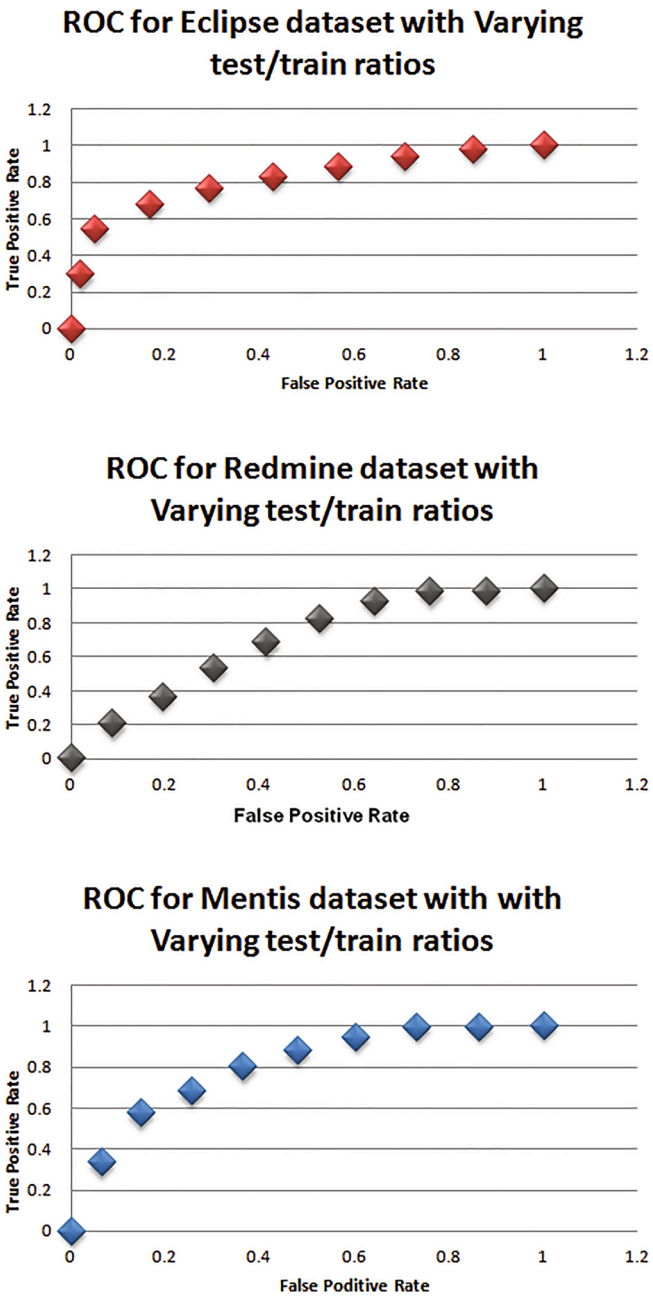


**Fig. 7.** ROC analysis with varying ratios of testing and training data for each dataset. Observations started from 90% training sets and gradually reduced by 10%, each time for all datasets, enabling us to observe the behavior. The correct and false cumulative calculations were conducted to produce TPR and FPR. The ROC curves exhibit almost identical behavior. However, Redmine shows a lower AUC compared to the other two datasets.

#### 4.5.2. Varying size of datasets

The dataset size is a crucial factor in evaluating model performance. To investigate effect of the size on the model, we carried out experiments with datasets containing varying numbers of records. *Preprocess* function takes summary and category labels as input and after stopword removal, stemming etc. provides the filtered data for further processing by Training.java class (Algorithm 1 Line # 3 to Line # 7). This filtered data is then stored separately for each bug category having its tokens, frequency and weight, thus training classes are obtained for each dataset chunk. i.e. for Data1000 (datset containing 1000 records) to Data100000 (dataset containing 1 million records) for Eclipse and also for Data1000 to Data300 of three different kinds of datasets (Redmine,

Mentis and Eclipse). The process of obtaining *BT* is explained in Section 3. After formation of separate class tables for each training set, the next step is to identify the key features. The function($l_i$) KeyFeatureIdentification ($T, W, f$) in FeatureExtraction.java class that takes token, frequency and weight of each bug and produces the key feature based on the weights as explained in Algorithm 1, Line # 20 to Line #33. Feature Identification process is explained in detail in Section 3.2.2. Training classes for different sized datasets is then fed into SPC($d_i, t_i, f_i, w_i$) function in SplHFBugs.java class to control the self-paced phases. for the first two, BT obtained earlier through BackTraceabilityDes.java class is used to map the highest and lowest weight tokens in each class as explained in Algorithm 1 Line # 35 to Line # 52. Finally, similarity dependent bugs are also classified using KFIM matrix. Steps are explained in Algorithm 2. Varied sized chunks of data goes through these steps separately. Finally, each chunk is evaluated for the results through EvaluationPRF.java class which contains all the formulas for calculating the four metrics used to evaluate the performance (i.e. precision, accuracy, recall and F1). In this class we have used original fractions obtained rather than converting them to percentage so that to make them fit for further graphical representations and that all the four metrics fit in the single scale.

The results so obtained are displayed in Fig. 8. The average precision is 95.4%, 93.3% and 85.3% for Eclipse, Redmine and Mentis respectively. Mentis obtains slightly less precision than the other two datasets. However, the sensitivity (recall) and F-1 score are still high – above 90% – and there is not much difference between these two parameters for all three datasets.

Word similarity between summary and bug category is quite low in the Mentis dataset, which leads to different number of easy samples in multiple testing chunks of data, which also contributes to the variation. However, only a few observations are missed or non-classified, which contributes to a greater recall after being processed by SplHFBugs.java. On the other hand, Eclipse contains a large number of classes, i.e., bug types, as compared to other two datasets, hence, different testing data contains different types, resulting in slightly lower recall and F-1 compared to other two datasets, since, the number of missed classes (bug types) is higher in eclipse. Yet, for Redmine and Eclipse, the proposed model's performance is quite high, and the two datasets did not behave drastically different with an increase in the number of records. Furthermore, Mentis also shows an average precision of 85%, which means that the classifier is capable of differentiating between the types, even with a small number of words contributing towards each bug type. Additionally, the recall for Mentis is higher and stable, whereas the F-score is also above 0.90 and closer to the rest, and it shows similar performance to the other two datasets.

In addition, to validate our hypothesis (RQ3), we also included results from the larger Eclipse dataset. Fig. 9 represents precision, accuracy, recall and F-Score with varying datasets sizes. Precision values increases from 60,000 records. However, the average is 96% and does not show drastic variations with 20,000 records added each time. the same applies to all other metrics. The bars in Fig. 9 further validate the results of the Eclipse dataset in Fig. 8, where the dataset size was smaller (1000 to 3000 records). This implies that even a large dataset size does not considerably affect the model performance, as the specific bug classes contains specific keywords. Hence, the defining features might not be affected by the size of the dataset. However, the amount of dependent and independent keywords and easier samples to be put in the self paced phases might have an impact on the model's performance.

The above statement is verified and validated further by various training samples, and the ROC curves in Fig. 7 and the 10-fold cross-validation curves for accuracy and precision, shown in Fig. 10, also suggest that the model is not affected by the dataset size. Therefore, our model is robust for data of various sizes.

### 4.6. *Validation and effect of self-paced induction*

To evaluate the self-paced learning effect of the model and test the hypothesis (RQ4), we conducted experiments with and without SPL induction. The comparison is illustrated in Fig. 10. The performance of our model is validated on different training sets. 10-fold cross-validation is used for this purpose (in which one portion of the dataset was left out for testing). Fig. 10 illustrates the accuracy and precision curve for 10-

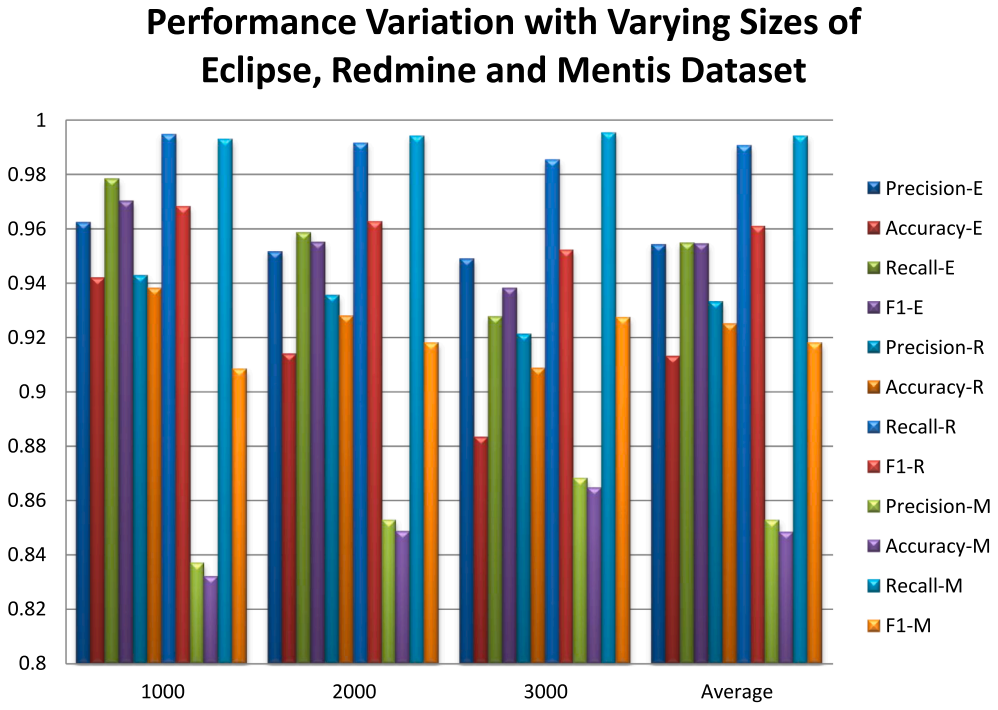


**Fig. 8.** Performance of the model on various size datasets. Testing and training sets are taken from the same data. Redmine depicts stable precision, whereas, Mentis shows lower precision than the other two datasets. Nonetheless, recall and F-1 are above 0.9.

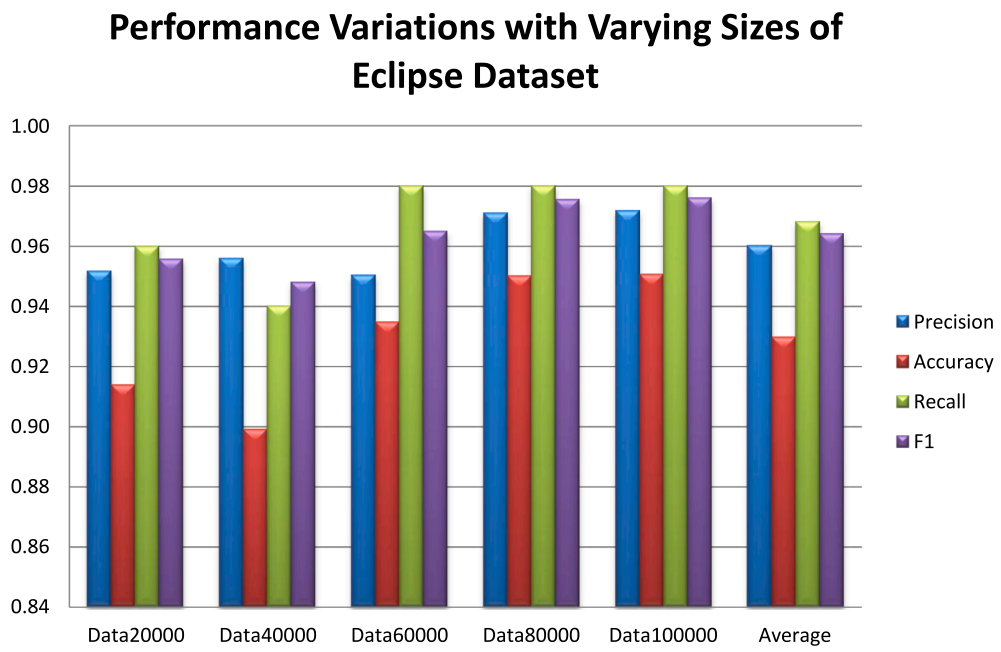


**Fig. 9.** Performance of the model on the various size Eclipse datasets. Precision remains stable with increasing dataset sizes, whereas recall and F-Score drops and become stable after 40,000 records.

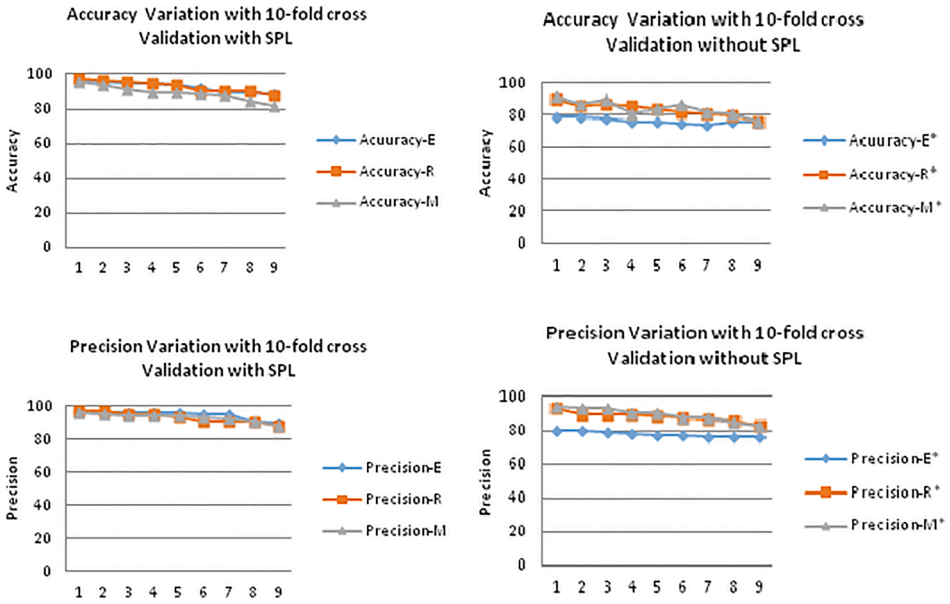


**Fig. 10.** Accuracy and precision of SPBC on 10-Fold cross-validation, with and without self-paced induction phase. The Eclipse and Redmine curves mimic each other to a substantial extent, whereas the Mentis curve lies a little low in accuracy when samples are incorporated in a self-paced fashion. Furthermore, the curves also show stability, and a few sharp fluctuations are observed compared to when these are tested without SPL stimulation. Accuracy and precision drop by 5% to 15% (on average) in the absence of the SPL phase. Mentis data exhibits a fluctuation of 5% on average, whereas this figure is 10% for Redmine and 15% for Eclipse.

fold cross-validation on different datasets. The Eclipse and Redmine curves mimic each other to a substantial extent, whereas the Mentis curve lies a little low in accuracy, when samples are incorporated in a self-paced fashion. Furthermore, the curves also show stability, i.e,. not many sharp fluctuations are observed when the experiment is conducted with SPL.

This is due to the fact that SPL makes the model more stable through the initial classification of a set amount of data samples, regardless of the dataset's nature. Moreover, the Mentis curve lies lower than the other two. This is because larger datasets provide more easy samples than smaller feature sets; thus, the model correctly maps more bugs in datasets with a larger number of features compared to those with a lower feature count. However, the lowest accuracy obtained is 89.01% for Mentis, which is still good, but slightly lower than Eclipse (94.91% precision) and Redmine (93.06% precision).

Eclipse and Redmine achieve 92% accuracy on average, while for the Mentis dataset, this figure dropped a little to 89% on average. Redmine and Eclipse produce almost the same results which proves that our model is stable and performance is not substantially affected by different natures of datasets. Furthermore, curves are smooth and exhibit less fluctuations.

On average, self-paced induction improves classifier accuracy by 14% in Eclipse, 9% in Redmine and up to 5% in Mentis. By contrast, without it, the models are prone to changing performance with changing data, which is reflected by the accuracy and precision curves without SPL. Mentis data exhibits a fluctuation of 5% on average, whereas

Redmine and Eclipse on average. One reason for these results is that the Eclipse data provides a large number of records that can be categorized as easy samples. As each class contains a larger number of frequent items, it has more data to feed into self-paced learner stages I and II. Therefore, more records are classified, meaning that performance is increased in this dataset. In comparison, other datasets make a smaller and less frequent bag of words, compared to the Eclipse data.

### 4.7. Analysis

From our results, it is evident that our model outperforms all other baseline and state-of-the-art methods. Compared to other methods, SPBC exhibits consistent performance in terms of Cohen's Kappa (0.96), better than the rest of the approaches in terms of performance. SPBC also maintained the stability, with only a 4% decrease in accuracy, compared to the other approaches, which showed a 12% to 35% decrease on varying datasets and 10-fold cross-validation. While other approaches seem to struggle here because of their text-dependent nature, our model also classified the non-similar bugs, especially in the Redmine dataset. Moreover, NNs also seem to be data-dependent, while for each dataset, separate tuning is required, and hence, performance varies for each input corpus. Accordingly, the model is capable of improving accuracy while also being stable, which will be quite helpful in automated bug-tracking systems. Furthermore, the model is robust for varying dataset sizes. The AUC values of 0.85 and 0.84 for the Eclipse and the Mentis datasets (respectively) with training ratios at a minimum of 10% show that the model is also robust and accurate in capturing unseen data. Redmine showed a decreased AUC (i.e., 0.74), but the model is capable of differentiating between types even with only 10% of data used for training and testing on 90% of the data. Furthermore, the model performance is also similar on varying sized and nature datasets with an average precision of 94.61% for Eclipse, 93.06% for Redmine and 93.43% for Mentis datasets. Furthermore, SPL improved accuracy by 14% in Eclipse, 9% in Redmine and up to 5% in Mentis. Without SPL, the models are prone to experience changing performance with changing data which is reflected by the accuracy and precision curves without SPL. In addition, the ANOVA results also prove that our approach significantly outperforms others. Hence, all of our research questions are validated.

To conclude, the integration of SPL with textually dependent and independent classification is found to be fruitful both for stability and improving the results. This validates our hypothesis that SPL brings robustness and effectiveness together. With this approach, our model outperforms all others in terms of accuracy, precision, F-measure, Recall and kappa. Based on our evaluation results, we can conclude that for short texts of special nature, both similarity and non-similarity should be considered simultaneously in order to improve the classifiers' performance, combined with SPL to control the data insertion and reduce errors. Finally, the stable and efficient classifiers will prove handy in the software industry in helping the developers to complete their tasks.

## 5. Threats to validity and limitations

Short descriptions of each bug are extracted from the Bugzilla, Mentis and Redmine bug-tracking repositories, and one is also taken from GitHub. The distribution of datasets is not uniform for all classes of bugs. The chances are high that the learning may be biased for a higher distribution class. This issue is addressed by taking random samples with varied distribution, which means that the model is still capable of identifying the bug types due to its text independent nature and can remove the bias to a significant extent.

Another threat to validity is that our model is tested on historical datasets, meaning that they may not be representative of runtime bugs. For mitigating this issue, we used the 10-fold cross-validation approach; moreover, unknown data is fed into the model, and the results so obtained are satisfactory (Section 4.4). Furthermore, we have selected popular bug-tracking system datasets with varying sizes and keywords to encompass possible real-time bugs in order to mitigate this threat. While the bugs are actual bug reports that were encountered, the labels are learned from known bug types, meaning that the model is likely to underperform in real scenarios. We plan that our future work will include real-time bugs to test our model exhaustively in order to determine its aptness in real-world applications with a minimum chance of error.

For SPL, decisions regarding easy and hard samples are human-oriented. For instance, we have used every possible definition of easy and hard sample, (e.g., most and least frequent tokens, highest and lowest weight tokens), and the sample definition yielding the best results is retained (i.e., highest and lowest weights were defined as easy samples and the rest as hard samples). The performance might vary for a different definition of easy and hard samples for different types of software systems.

## 6. Conclusions and future work

This research work presents a novel bug classifier to solve the categorization problem of bug reports on data of varying aspects and sizes. The framework presents an SPL model designed to identify bug types in historical bug reports of open-source projects by classifying these bug reports into categories. These can further be used to speed up bug resolution and correct assignment by pertinent developers with germane expertise.

The SPBC model makes use of a back-traceability matrix to assign data samples in a controlled fashion, hence exploiting the natural phenomena of learning from easy to hard concepts. The model first categorizes those bugs that are marked by the classifier as easy samples on the basis of the feature set and weighted matrix. After every easier sample is mapped, the classifier then learns the defining features of each bug class and ultimately assigns a class to every bug.The SPBC framework provides significant performance improvement in terms of robustness and stability, AUC and kappa measures. Experiments were also conducted to study the importance of the self-paced induction of data samples. The results clearly indicate that performance is improved to a considerable extent (5% to 15% on average) when samples are introduced in a controlled manner. The results also reveal that our model outperforms the existing approaches in terms of performance and stability, and there is also a substantial improvement in results with SPL on 10-fold cross-validation. Our model exhibits consistent behavior and is stable (an average 4% drop in precision on different datasets in the 10-fold cross-validation) compared to the other baseline methods (up to a 30% drop in precision). .

The spatial relation of data with software systems is also of great importance. Our research supports the conclusion that substantially different results can be obtained when data is introduced into a system in a systematic manner. Keeping this in mind, a potential avenue for future work would be to study the effect of SPL for runtime bug identification, instead of historical bug reports. This can help in improving the system more, by understanding the exact behavior in the real deployment of intelligent systems in industry; hence, it could help in gaining the trust of practitioners to rely on automated systems for bug fixing and triage. Another future work avenue would be to apply this approach in combination with deep learning methods to the field of natural language processing, which has huge room for improvement and has not yet been tested with self-paced regularizers for this problem.

The practical implications of deep learning in software debug recommendation systems, bug triage systems and bug fix systems have not yet been explored. In particular, the impact of SPL combined with the deep learning methods is still unexplored. Thus, an interesting avenue of future work would be to study the impacts of an integrated deep self-paced approach in improving the expert systems. Also, this research will be extended to build a self-paced debug recommendation system capable of learning data from multiple domains and building its

own easy and hard samples to learn gradually from simple to complex bug fix solutions from heterogeneous sources. This will help in a building a better triage system that encompasses extensive information from a wider range of sources.

## CRediT authorship contribution statement

**Hufsa Mohsin:** Conceptualization, Methodology, Formal analysis, Investigation, Data curation, Writing - original draft. **Chongyang Shi:** Conceptualization, Methodology, Investigation, Data curation, Writing - review & editing, Supervision.

## Declaration of Competing Interest

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

## Acknowledgements

This work was supported by the National Key Research and Development Program of China [No. 2018YFB1003903]; National Natural Science Foundation of China [No. 61502033, 61472034, 61772071, 61272361 and 61672098]; and International Graduate Exchange Program of Beijing Institute of Technology, under Chinese Government Scholarship [CSC No. 2017GBJ008248].

## References

Aggarwal, C. & Zhai, C. (2012). A Survey of Text Classification Algorithms. In Z. C. Aggarwal & C. (Ed.), Mining text data (pp. 163-222). Boston, MA: Springer. doi: 10.1007/978-1-4614-3223-4_6.

Allahyari, M., Pouriyeh, S., Assefi, M., Safaei, S., Trippe, E. D., Gutierrez, J. B. & Kochut, K. (2017). A brief survey of text mining: Classification, clustering and extraction techniques. Big data analytics as a service – bigdas, Transactions on Knowledge and Data discovery, abs/1707.02919. Halifax, canada. Retrieved from http://arxiv.org/abs/1707.02919.

Almhana, R., Mkaouer, W., Kessentini, M. & Ouni, A. (2016). Recommending relevant classes for bug reports using multi-objective search. Proceedings of the 31st IEEE/ACM international conference on automated software engineering (pp. 286–295). Singapore: ACM. doi:10.1145/2970276.2970344.

Amensisa, A. D., Agrawal, P., & Patil, S. (2018). A survey on text document categorization using enhanced sentence vector space model and bi-gram text representation model based on novel fusion techniques. In *Proceedings of the 2nd international conference on inventive systems and control-ICISC* (pp. 218–225).

Anvik, J. & Murphy, G.C. (2011). Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology, 20(3), 10:1–10:35. doi:10.1145/2000791.2000794.

Cao, L. (2016). Non-IID recommender systems: A review and framework of recommendation paradigm shifting. *Journal of Engineering, 2*(2), 212–224.

Chen, D., Li, B., Zhou, C. & Zhu, X. (Feburary, 2019). Automatically Identifying Bug Entities and Relations for Bug Analysis. 1st International workshop on intelligent Bug Fixing coRelated with SANER'19. Hangzhou, Zhejiang, China: IEEE.

D'Ambros, M., & Lanza, M. (2012). An extensive comparison of bug prediction approaches. *Journal of Empirical Software Engineering, 17*(4–5), 531–577.

Elmishali, A., Stern, R., & Kalech, M. (2018). An Artificial Intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence, 69*, 147–156.

Fan, Y., Tian, F., Qin, T., Bian, J. & Liu, T. -Y. (2017). Learning What Data to Learn. CoRR, abs/1702.08635, arXiv. Retrieved from http://arxiv.org/abs/1702.08635.

Guzman-Cabrera, R., Montes-y-Gomez, M., Rosso, P., & Pineda, L. V. (2009). Using the web as corpus for self-training text categorization. *Journal of Information Retrieval, 12*(3), 400–415. https://doi.org/10.1007/s10791-008-9083-7.

Hattori, L., Jr., G.P., Cardodo, F. & Sampaio, M. C. (2009). Mining software repositories for software change impact analysis: A case study. 8th Proceedings of the 23rd Brazilian symposium on Databases, (pp. 210–223). Brazil.

Jiang, M., Liang, Y., Feng, X., Fan, X., Pei, Z., Xue, Y., & Guan, R. (2018). Text classification based on deep belief network and softmax regression. *Neural Computing and Application, 29*(1), 61–70. https://doi.org/10.1007/s00521-016-2401-x.

Jie, Z., xiaoyin, W., Dan, H., Bing, X., Z. & Hong, M. (2015). A survey on bug-report analysis. Science China Information Sciences, 58(2), 1–24. doi:10.1007/s11432-014-5241-2.

Kalyanasundaram, S., & Murphy, G. C. (2012). Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India software engineering conference (ISEC)* (pp. 125–130). Kanpur, India: ACM. doi:.1145/2134254.2134276.

Kim, S. & Whitehead, E .J. (2006). How long did it take to fix bugs? Proceedings of the 2006 international workshop on mining software repositories (MSR'06) (pp. 173–174). NewYork: ACM. doi:10.1145/1137983.1138027.

Li, X., Jiang, H., Liu, D., Ren, Z., & Li, G. (2018). Unsupervised deep bug report summarization. In *Proceedings of the 26th international conference on program comprehension (ICPC-18)* (pp. 144–155). Gothenberg, Sweden: IEEE.

Li, C., Wei, F., Yan, J., Zhang, X., Liu, Q., & Zha, H. (2018). A self-paced regularization framework for multi-label learning. *IEEE Transactions on Neural Networks and Learning Systems, 29*, 2660–2666.

Limsettho, N., Hata, H., Monden, A., & Matsumoto, K. (2016). Unsupervised bug report categorization using clustering and labeling algorithm. *International Journal of Software Engineering and Knowledge Engineering, 26*(7), 1027–1054. https://doi.org/10.1142/S0218194016500352.

Lu, J., Behbood, V., Hao, P., Zuo, H., Xue, S., & Zhang, G. (2015). Transfer learning using computational intelligence: A survey. *Knowledge-Based Systems, 80*, 14–23. https://doi.org/10.1016/j.knosys.2015.01.010.

Lu, Z., Zhu, Y., Pan, S. J., Xiang, E. W., Wang, Y., & Yang, Q. (2014). Source free transfer learning for text classification. In *Twenty-eighth AAAI conference on artificial intelligence* (pp. 122–128). Québec, Canada: AAAI press. https://doi.org/10.1002/smr.1770.

Ma, F., Meng, D., Xie, Q., Li, Z. & Dong, X. (2017). Self-paced co-training. Proceedings of the 34th international conference on machine learning (ICML) 06-11 August 2017, 70, pp. 2275–2284. Sydney, Australia.

Neelofar, Javed, M. Y. & Mohsin, H. (2012). An automated approach for software bug classification. Proceedings of the 2012 sixth international conference on complex, intelligent, and software intensive systems (CISIS), July 04–06, 2012 (pp. 414–419). Italy: IEEE, Computer Society. doi:10.1109/CISIS.2012.132.

Pe, T., Li, X., Zhang, Z., Meng, D., Wu, F., JunXiang, & Zhuang, Y. (2016). Self-paced boost learning for classification. Proceedings of 25th international joint conference on artificial intelligence IJCAI-16 (pp. 1932–1938). NewYork, USA: IJCAI, AAAI Press.

Qiu, J., Wu, Q., Ding, G., Xu, Y. & Feng, S. (2017). A survey of machine learning for big data processing. Journal on Advances in Signal Processing, EURASIP, pp. 1–16.

Razzaq, S., Li, Y., Lin, C. & Xie, M. (2018). A study of the extraction of bug judgment and correction times from open source software bug logs. IEEE 18th international conference on software quality, reliability and security companion (pp. 229–234). Lisbon, Portugal: IEEE, ISBN:978-1-5386-7839-8.

Shivaji, S. & E., J. (2013). Reducing features to improve bug prediction. Transactions on Software Engineering, 39(4), 552–569.

Thung, F., Lo, D., & Jiang, L. (2012). Automatic defect categorization. In *Proceedings of the 19th international working conference on reverse engineering* (pp. 205–214). Kingston, Canada: IEEE, Computer Society. https://doi.org/10.1109/WCRE.2012.30.

Thung, F., Shaowei, W., David, L. & Jiang, L. (2012). An empirical study of bugs in machine learning systems. Proceedings of the 23rd IEEE international symposium on software reliability engineering – ISSRE 27-30 November 2012 (pp. 271–280). Dallas: IEEE, Computer Society. doi:10.1109/ISSRE.2012.22.

Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. In *Proceedings of the 38th international conference on software engineering* (pp. 297–308). Australia: ACM. https://doi.org/10.1145/2884781.2884804.

Xiaobing Sun, W. Z. (2019). Bug localization for version issues with defect patterns. *IEEE Access, 7*, 18811–18820. https://doi.org/10.1109/ACCESS.2019.2894976.

Xu, C., Tao, D., & Xu, C. (2015). Multi-view self-paced learning for clustering. In *Proceedings of the twenty-fourth international joint conference on artificial intelligence (IJCAI)* (pp. 3974–3980). Buenos Aires, Argentina: AAAI Press.

Yu, L., Kong, C., Xu, L., Zaho, J. & Zhang, H. (2008). Mining bug classifier and debug strategy association rules for web-based applications. Proceedings of the fourth international conference on advanced ddata mining applications, ADMA. LNAI 5139, pp. 427–434. Berlin Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-88192-6_40.

Zaidman, A., Rompaey, B. V., Demeyer, S. & Deursen, A. V. (2008). Mining software repositories to study co-evolution of production and test code. 1st International conference on software testing(ICST' 08) (pp. 220–229). IEEE Xplore. doi:10.1109/ICST.2008.47.

Zhang, H. & Zhong, G. (2016). Improving short text classification by learning vector representation of both words and hidden topics, 102, pp. 1–140. Knowledge-Based Systems, 102, pp. 76–86. doi:10.1016/j.knosys.2016.03.027.

Zhang, T., Jiang, H., Luo, X., & Chan, A. T. (2016). A literature review of research in bug resolution: Tasks, challenges and future directions. *The Computer Journal, 59*(5), 741–773. https://doi.org/10.1093/comjnl/bxv114.

Zhang, X., Yao, Y., Wang, Y., Xu, F. & Lu, J. (2017). Exploring metadata in bug reports for bug localization. 24th Asia-Pacific software engineering conference 'APSection 2017 (pp. 328–337). Nanjing, China: IEEE, Computer Society. doi:10.1109/APSection 2017.39.

Zhao, Q., Meng, D., Jiang, L., Xie, Q., Xu, Z., & Hauptmann, A. G. (2016). Self-paced learning for matrix factorization. In *29th AAAI conference on artificial intelligence* (pp. 3196–3202). Texas, USA: AAAI Publications.

Zheng, T., & Wang, L. (2018). Unlabeled text classification optimization algorithm based on active self-paced learning. In *IEEE international conference on big data and smart computing (BigComp)* (pp. 404–409). Shanghai, China: IEEE Computer Society. https://doi.org/10.1109/BigComp.2018.00066.

Zhou, Y., Tong, Y. X., Gu, R. H., & Gall, H. (2016). Combining text mining and data mining for bug report classification. *Journal of Software Evolution and Process, 28*(3), 150–176. https://doi.org/10.1002/smr.1770.

Zimmermann, T., Nagappan, N., & Philip, J. G. (2012). Characterizing and predicting which bugs get reopened. In *Proceedings of the thirty-fourth international conference on software engineering (ICSE)* (pp. 1074–1083). New Jersy, USA: IEEE Press.

Zhang, W., Wang, S., & Wang, Q. (2016). KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity. *Information and Software Technology, 70*, 68–84. https://doi.org/10.1016/j.infsof.2015.10.004.