Knowledge-Based Systems 236 (2022) 107711

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

SPAN: A self-paced association augmentation and node embedding-based model for software bug classification and assignment

Hufsa Mohsin [a], Chongyang Shi [a,*], Shufeng Hao [b], He Jiang [c]

[a] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
[b] College of Data Sciences, Taiyuan University of Technology, Shanxi 030024, China
[c] School of Software, Dalian University of Technology, Dalian, Liaoning 116024, China

ARTICLE INFO

Article history:
Received 25 March 2021
Received in revised form 5 November 2021
Accepted 7 November 2021
Available online 26 November 2021

Keywords:
Bug classification
Bug triage
Association augmentation
Graph embedding
Bug assignment
Bug report analysis

ABSTRACT

Effective bug classification and assignment to relevant developers improves the efficiency of software management. However, textually dependent approaches produce inconsistent results on varying datasets, while approaches that depend upon multi-source data can produce dataset conflicts and inaccuracy. Accordingly, we introduce a model based on **S**elf-**P**aced **A**ssociation augmentation and **N**ode embedding (**SPAN**), which uses an effective combination of textually dependent and independent bug categorization to produce consistent results, followed by a bug assignment mechanism to prevent conflicts. To this end, we present a novel unified classifier and assignment model that exploits the connections between nodes in the **S**oftware **B**ug **R**eport **Net**work (**SBRNet**) to identify the target features. The model is capable of accurately categorizing bugs in a self-paced manner with association augmentation. Finally, we present an approach that assigns the most appropriate developer for bug resolution through SBRNet node information embedding. Our deep two-step self-paced solution is capable of categorizing software bugs with improved accuracy, while still utilizing fewer features. Results reveal that our model is more effective (up to 98% classification accuracy and 96% for bug assignment) when compared to its counterparts.

© 2021 Elsevier B.V. All rights reserved.

1. Introduction

Bug report analysis has attracted a great deal of interest from developers over the past few years, as it provides basic information about software malfunctions or unusual behavior. Bug classification and triage have long been fundamental problems faced by developers. Issues with natural language processing, technical information, and the unconstrained format of reporting have made bug classification and assignment into tedious tasks. Effective classification and bug triage promotes better resolution, while in turn also aiding the user and system entities, by making the system error-free. Several attempts [1–7] have been made to solve bug categorization and assignment problems from various perspectives, ranging from a simple machine learning-based approach by Cubranic and Murphy [8], Neelofar et al. [5] to semi-supervised (Xuan et al. [9]) and fully automated [10], POS tagging [11], ensembles [12], transfer learning (Liu et al. [13]) and advanced self-paced learning (SPL) approaches [4]. Each of these methods has attempted to resolve issues with or improve the efficiency of existing approaches.

Bug report categorization forms part of the major research effort aiming to deal with attribute similarity between pieces of information in a bug report, as a basic assumption. For instance, [5,6,14–16] presented bug localization, automated summary generation, source code defect prediction, and bug triage respectively, with bug report classification representing only a small subpart of their work. Some researchers, however, have extensively reported on bug report classification problems as a separate issue; for example, Limsettho et al. [10] proposed a bug categorization framework based on textual similarity that does not require labeled data. Neelofar et al. [5] also presented a Naive Bayes classifier for bug report classification. Machine learning algorithms widely used for classification include SVM, Naive Bayes, Decision Tree, and Neural Networks [17–19]. LDA topic modeling [10,20] has also been used to determine feature similarity in the bug classification context. However, topic modeling techniques [13,21] are the most common approaches in the bug assignment and triage context. Tossing graphs [3] have also been introduced to address the incorrect assignment

* Corresponding author.
E-mail addresses: hufsa.bit@yahoo.com, hufsa.mohsin@comsats.edu.pk (H. Mohsin), cy_shi@bit.edu.cn (C. Shi), haoshufeng@tyut.edu.cn (S. Hao), jianghe@dlut.edu.cn (H. Jiang).

https://doi.org/10.1016/j.knosys.2021.107711
0950-7051/© 2021 Elsevier B.V. All rights reserved.

of bug reports to developers. The developer expertise model [1] and portfolio management for online recommendation through transfer learning have also been proposed as a means of achieving developer assignment to a particular bug, as presented by Liu et al. [13].

Machine learning approaches and information retrieval techniques are prevalent in the field of bug classification and assignment; however, the current literature presents either textually dependent or fully concept-based approaches that lack stability on variable datasets. Hence, classification and tagging tasks for the most part are performed manually by practitioners. Similarly, for bug triage, several different features of a bug report are utilized: these include the source code (for vocabulary-based approaches [1]), resolution history, summary, description, priority, CC list, temporal data, and so on. These diverse features lead to conflicting results, particularly when a solution is attempted through hybrid approaches. Moreover, data-reduction (Zou et al. [12]) and instance selection techniques have been proposed for improving bug triage; however, high efficiency has not yet manifested, none of these approaches can achieve greater precision and manual intervention by developers is still prevalent. As a result, further work is required to achieve improvements in this area.

Our present work is motivated by current applications of graphs and networks in the software engineering domain [22]. A knowledge graph is composed of multiple relations among entities and edges [23]. Organizing data in knowledge graph form is known to facilitate rapid problem identification and a better understanding of information flow among various nodes; thus, it is a basis of many recommender systems [24–26] and link prediction models [27]. Knowledge in a graph can be expressed in the form (head, relation, tail) [28]. On this basis we identified all possible relations between a bug report and other entities, and accordingly constructed a **S**oftware **B**ug **R**eport **Net**work (**SBRNet**). SBRNet has proven to be useful for the prompt identification of features that can be utilized for bug labeling and assignment solutions. It also helps in identifying which fields are irrelevant, and thus reduces the chances of conflict among various data features, achieving respectable results. The work most relevant to ours in this context, first presented by Chen et al. [29], achieves the automatic category identification of bug entities and relations in bug reports. However, these authors defined a total of eight types of bug relations by incorporating the recurrent neural networks (RNNs) with a dependency parser. However, our work assigns each bug to its relevant category by identifying different tags for each bug. Similarly, Zhou [30], presented a construction framework for data organization using knowledge graphs from multi-source data, proposed principles and methods for identifying bug entities and relationships, and further constructed a preliminary knowledge graph based on the bug repository. Our model differs from [30] in terms of both the problem and solution context. We constructed SBRNet for bug type classification with a major focus on the descriptive field entity of a bug report. The main focus of the knowledge graphs outlined in [30] is on the code context of a bug report – **Q**uestion & **A**nswer (**Q&A**) – to fix a bug. Furthermore, the bug-specific entities in [30] are classified into "*component*", "*specific*" and "*general*" categories, which represents another difference from our approach.

In particular, although [29,30] serve as a basis and motivation for improving the performance of current classifiers, these works have not addressed the problems of classification or bug triage in any depth; instead, their primary focus was on knowledge graph construction with named entity identification. To incorporate the graph knowledge into bug categorization and assignment, and to ensure we can enjoy the benefits of knowledge graph, we exploit the connections with the developer and the descriptive field entities of a bug report.

To this end, we propose a unified model for solving the bug classification and assignment problems. This two-stage model first classifies the bugs into categories, then assigns the developers to a particular type of classified bug through the utilization of SBRNet information. Our novel approach offers the benefits of improved accuracy and stability while having fewer features to investigate. The token association-based classifier automatically classifies bugs into their corresponding type using self-paced, induced data by examining the descriptive fields of SBRNet. Unlike other existing classifiers, a self-paced classifier helps in maintaining both stability and access in contact [4]. These accurately classified bugs are then assigned to the developers by analyzing only the predefined entity of SBRNet; i.e., the model is trained on the information of a developer who has already fixed bugs of a particular type.

The model works by exploiting the relationships between a bug report and its corresponding entities. These relationships are then incorporated as node information of the developer entity and token association augmentation of the descriptive field entity of SBRNet. The model provides data/feature reduction and focuses on bug classification as an initial step, after which it utilizes the identified categories to map a particular bug type to a relevant developer in SBRNet. The major contributions of our model are as follows:

- The proposed model presents a novel representation of the bug report life cycle, named SBRNet, to identify the entities and features required to solve the classification and assignment problem for software bug reports.
- The proposed model presents a novel deep self-paced bug classifier, based on open-source software projects, which assigns bugs to relevant developers with a reduced feature set by introducing a novel embedding of token associations and node information embedding from SBRNet, in a controlled and self-paced manner.
- The proposed bug classifier achieves an average of 98% and 96% assignment accuracy on a unified dataset obtained from Eclipse, Mentis, Redmine, and various other platform bugs of Eclipse.

2. Motivation

We believe that no task stands on its own; rather, every piece of information forms a network of interconnected entities and attributes. Similarly, bug reports also make up a closed network comprising software, users, developers, and other artifacts. In this way, we aim to employ the graph representation of bug reports for effective bug classification and assignment. The internal and

Table 1
Symbol table.

Symbol	Description
BTS	Bug Tracking System
S	Software
SA	Software Artifact
B_r	Bug report
U	Set of users
U_n	N number of users
U_e	End users
U_{en}	N number of end users
D	Developers
T	Set of tokens
d_f	Descriptive entities
p_f	Predefined entities
a_f	Attachment entities
M_c	Mode of communication
bl	Blogs

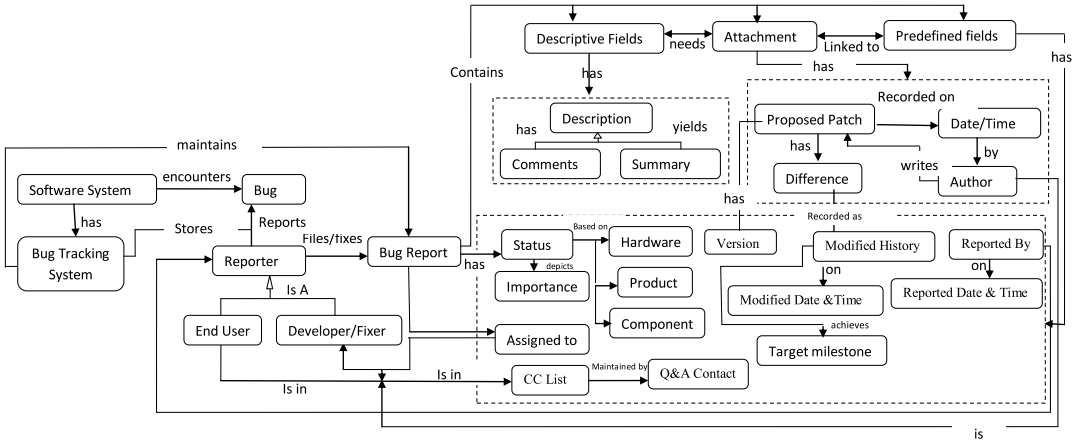

Fig. 1. Software bug report graph.

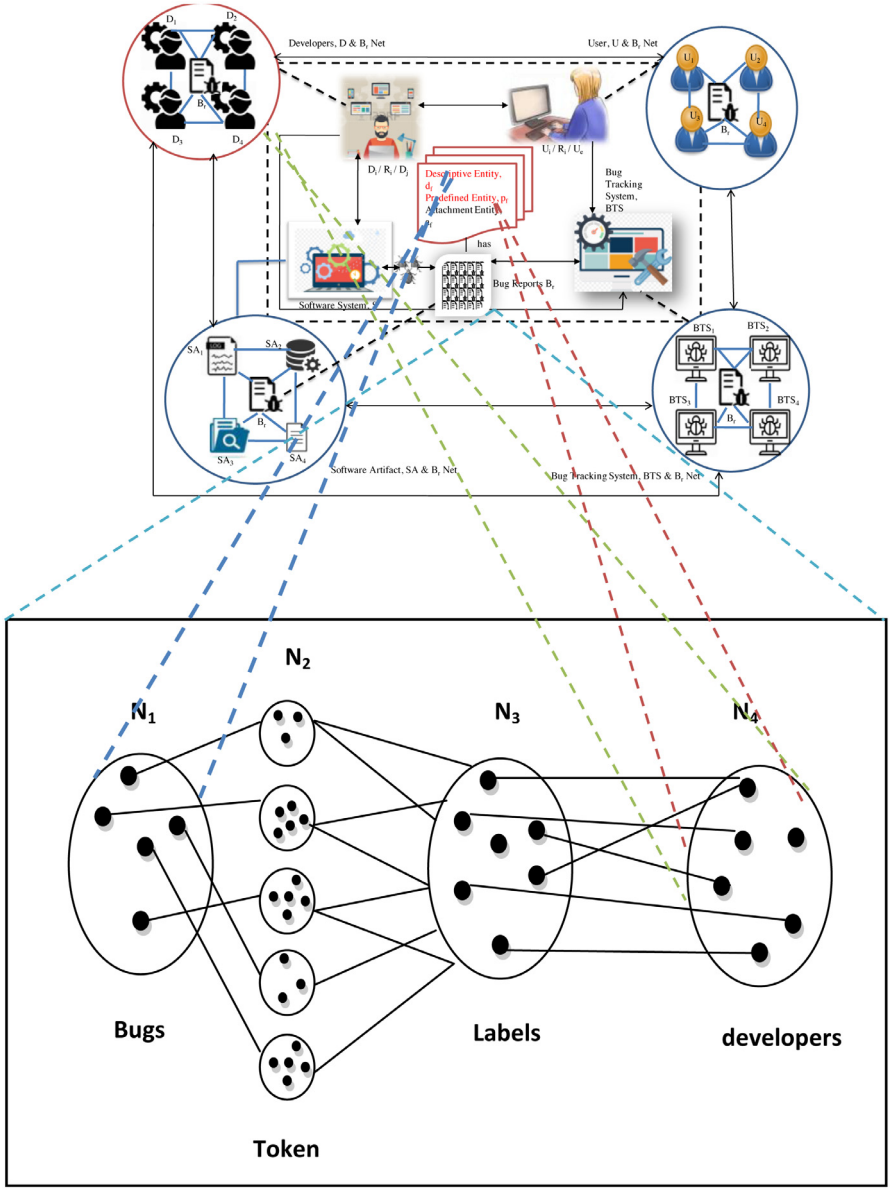

Fig. 2. Software bug report network and node graph.

Table 2
Software bug report sample summary and developers mapping.

Developer	Summary
mdelder	[CommonNavigator] Cannot Drop an External File into Project Explorer (Mac and Linux)
mdelder	[Wizards] Wizard progress label not cleared between runnables
michaelvanmeekeren	[Workbench] Use of hard coded font spews warnings on some platforms
michaelvanmeekeren	[Perspectives] Perspective bar incorrectly updated when opening new perspective
michaelvanmeekeren	[Preferences] Export should prompt for save before exporting
michaelvanmeekeren	[IDE] Deleting a linked folder can delete folder contents when read only files are involved

Table 3
Software bug report sample bug types and developer mapping.

Developer	Summary
mdelder	CommonNavigator
mdelder	Wizards
michaelvanmeekeren	Workbench
michaelvanmeekeren	Perspectives
michaelvanmeekeren	Preferences
michaelvanmeekeren	IDE

overall connections between the various entities of a general Eclipse bug report are presented in Figs. 1 and 2 respectively. Other bug tracking systems also store information into a similar type of field, although there are small difference in terminologies and grouping. The Eclipse software is widely used in research and has a very well-defined bug maintenance process, which is why we have constructed its graph. As shown in Fig. 1, software systems record bugs with the help of a bug tracking system. Bugs encountered across the entire software life cycle are either reported by a user or by a developer who creates a bug report. A bug report in general, and Eclipse bug reports in particular, stores the knowledge in the form of three basic entities: (1) descriptive fields (e.g., summary, description, and comments), (2) predefined (e.g., product, priority, severity, and resolution status), and (3) attachment fields (e.g., screenshots, code snippets, patches). This information is typically either provided by the user at the time of reporting or added by the developer. As is also clear from Fig. 1, these three entities/fields are further connected through multiple attributes and sub-attributes. Hence, the greater the number of attributes used in a solution, the more complicated it is to preserve the connections and avoid conflicts in flow. Thus, we formally define SBRNet as follows:

An SBRNet is a closed network initiated by a B_r reported through a Bug Tracking System (BTS) by reporter $R_i \in U_i$, D_i, U_{ei} to a developer D_i for resolution, who either fixes the b_i himself or delegates it to the relevant D_j to be fixed. D_j responds back to the reporter through M_c, forming a closed loop.

The symbols employed in SBRNet, along with their definitions, are listed in Table 1. Moreover, Fig. 2, illustrates a general presentation of the connections between each entity throughout each field. Every bug report, B_r is evaluated by a developer $D \in \{D_1, D_2, \ldots, D_n\}$, or sent to the relevant D_j for resolution. A closed Bug Report Network is formed by sending the response back to the initiating reporter R_i through a reporting medium entity M_c; here, M_c is the mode of communication, which can be blogs (bl), email, Instant Message (IM) or Question & Answer sessions (QA).

Here, we are concerned primarily with the connection between the bug report B_r and assigned developer D_i through the descriptive and predefined fields of B_r. Similar to the COME+ node embedding model [31], rather than attempting to analyze several predefined fields (p_r) of a bug report, our objective is to analyze the descriptive field d_r (community) of the network, i.e. the summary. The summary is first analyzed to categorize the bug report into different types, after which a particular type is assigned to the developer D_i. The $D_i \in p_r$ are contained in the assigned field of the dataset. Thus, we used four nodes – namely, Bugs, Tokens, Labels, and Developers – from the SBRNet depicted in Fig. 2. Let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of n bugs reported by the users, U. Each bug b_i has its summary or short description containing a set of tokens $T = \{t_{i1}, t_{i2}, \ldots, t_{in}\}$, for the ith bug. These tokens form associations with other bugs in set B, thus forming a network of different combinations with various b_i. These associations in turn yield the classification of bugs of different types, i.e. labels $L = \{l_1, l_2, \ldots, l_n\}$. A bug b_i can contain one or more than one label. The set L_i is then used to predict the developers D_i from the set of various developers D; it is possible that two developers may be assigned different bugs with the same label, and also that there may be no developer capable of being assigned to a particular labeled bug. The formal definition of SBRNet is as presented below, while the symbols utilized and their definition are listed in Table 1.

Accordingly, we embedded the vertex/node-based graph information at two stages of the model. The first stage focuses on the bug summary and tokens, along with associations, which produce bug labels that act as nodes in the next stage in order to predict the developers for bug assignment. The model is first trained on the already assigned developers capable of fixing the bug and then validated on the unseen dataset. An example bug report for Eclipse is provided in Table 2. For triage, we will only consider the connection marked in red in Fig. 2; a sample description is provided in Table 2. The summary provided in Table 2 is first preprocessed and categorized into its corresponding type, which will then be mapped to the relevant developer to yield Table 3. Note that this process constitutes very few attributes of bug reports (i.e., it contains Summary and Developer information only), meaning that it reduces the feature set by a considerable amount. Furthermore, it is easier to map assigned data types to the relevant developer than to match the summary and developer expertise.

3. Related works

In recent years, diverse methods and techniques for software bug triage have been developed and tested which range from simple machine learning to transfer learning. The subsections below present a summary of relevant approaches related to the problem under consideration.

3.1. Bug assignment and triage approaches

Triage is viewed as a classification problem by Cubranic and Murphy [8]. Naive Bayes, the classic text classification algorithm, is applied to the summary, free-text description, assigned to, status, and CC list of Eclipse bug reports by these authors. However, being one of the earliest works aiming to assign bug reports to developers, the results were not very accurate; the accuracy was quite low at only 30%. Their work was further improved by Xuan et al. [9], who developed a semi-supervised machine learning approach for bug report labeling by utilizing Naive Bayes and the expectation–maximization algorithm to assign a specific report to the relevant developer. These authors used the short and first long description field of the Eclipse bug reports are used

for bug labeling, while the *assigned-to* and *status* fields are used for assignment to developers. This approach improved the Naive Bayes classification accuracy by up to 6%. The bug triage problem was therefore solved by conceptualizing it as a classification task. Anvik and Murphy [32], further enhanced the performance of machine learning recommenders and attained higher precision by employing SVM, Naive Bayes, and C4.5, and found that SVM obtains higher accuracy on the Eclipse and Firefox datasets when recommending a list of relevant developers to fix the bug reports. This approach achieved precision ranging from 70% to 98% on five open-source projects. Zou et al. [12], propose a feature and instance selection technique for data-set reduction through a bag-of-words approach, thereby achieving 5% improvement in Naive Bayes results.

All these initial baseline methods employ machine learning algorithms that are normally text-dependent, while multiple fields of a bug report are also utilized. A lack of stability on varied datasets from different projects can also be observed. By contrast, our approach is tested on a mixed blend of datasets from multiple projects and achieves far more stable and consistent performance. Moreover, even though only two attributes are utilized, reasonable performance is still achieved.

Aside from machine learning, the other prevalent approach in bug triage/ assignment is the probabilistic graph-based model (i.e., tossing graph). Jeong et al. [7], propose a Markov-chain-based graph model to generate tossing graph history. Experiments on Eclipse and Mozilla datasets reveal improved results for Naive Bayes and Bayesian classifiers. Bhattacharya et al. [33] present another tossing-graph-based model for bug triage that leverages both machine learning tools and tossing graphs. Keywords generated from the bug report title, summary and description, temporal information, and assigned developer fields, are utilized to predict developers capable of fixing the bug. Naive Bayes coupled with product-component, tossing graph, and incremental learning was found to achieve up to 86% accuracy on the Eclipse and Mozilla datasets. Another fine-grained and multi-feature tossing graph is proposed in [34].

Tossing graphs are typically used to identify the incorrect or accidental assignment of bug reports to developers by assigning and reassigning bug reports to multiple developers. However, this developer assignment involves the use of machine learning tools are also impacted by from the same problems described above. Furthermore, for several features and larger datasets, the length of the graphs is also a major concern.

Analyzing developer expertise to conduct bug triage is also considered as a means of facilitating the efficient assignment of bugs to the relevant developers. DREX [2] is a k-nearest neighbor-based approach for recommending the developer best suited to resolving a bug based on his expertise. Mozilla Firefox reports are selected for experimentation, while Mozilla and Eclipse reports were used to validate the approach. Matter et al. [1] presented a vocabulary search-based expertise model that identifies relevant text in source code and bug reports in order to assign a particular developer to fix the bug. Final results are obtained through an Eclipse case study by comparing the predicted developers with actual developers, and a 33.61% top-1 precision is reported. Facilitating bug triage with data from multiple sources was attempted by Liu et al. [13]. The two-stage approach (offline and online learning) works on a similarity measure and topic modeling for word count, along with an assessment of possible topics from multiple data sources of different formats and nature, to create a portfolio theory that can be used to provide fixer recommendations while considering accuracy, cost, and time optimization. Finally, 86.09% accuracy is attained.

Expertise-based models require multiple features and data sources to obtain a highly accurate base capable of predicting the most appropriate developer. Furthermore, data of multiple types tends to lead to conflict or incompatibility among the approaches used for classification. To resolve these issues, our model provides a simplified approach that utilizes few features, while still attaining better results.

3.2. Bug classification and self-paced approaches

This section highlights the research related to software bug classification and the application of self-paced learning (SPL). Bug classification is often not dealt with as a distinct problem; rather, it is considered part of a solution to a bigger problem, such as triage or localization. However, since we assert herein that effective classification leads to efficient bug triage, some bug classification works are discussed below, along with several applications of SPL in this particular field.

Bug classification techniques are generally textually dependent and work by matching the words in a data corpus. For instance, Somasundaram and Murphy [15] conducted a study to detect bug misclassification by applying both text mining and data mining techniques to bug reporting data in order for automation of the prediction process. The aspect of their work that relates most to ours is the part that leverages text-mining techniques to analyze the summary of bug reports, then classifies them into three levels of probability: bugs, non-bugs, and average bugs. The authors utilized Bayesian networks and multinomial Naive Bayes. Xuan et al. [9] presents a semi-automated approach for bug labeling and classification of Eclipse reports for the bug triage task, employing the Naive Bayes and expectation–maximization approaches along with weighting schemes to boost performance. Neelofar et al. [5] presented a Naive Bayes classifier for bug report classification. Other existing text classification approaches also depend on lexical similarity. For example, [35] devised a semi-automated lexical approach for the language-independent classification of specific non-English web corpora. Moving from supervised to unsupervised classification of bug reports, Limsettho et al. Limsettho et al. [10] propose a topic modeling framework for bug categorization. Word tokens extracted from the description, summary, and title undergo topic modeling to yield topic membership vectors, which are then ranked to obtain categories through Hierarchical Dirichlet Allocation (HDA). Part-of-speech-based tagging is also common for labeling different types of documents. Its effect on bug reports was studied in [11]. Tian and Lo [11] conducted a survey on POS-tagging-based approaches used in bug reports and compared their performance on normal English documents relative to bug reports. The findings show that the models perform differently on bug reports due to the existence of multiple challenges other than natural language therefore, the performance on these reports decreased by 10% on average relative to normal documents.

Some relevant approaches are the word vector- and co-occurrence-based techniques. Cubranic and Murphy [8], for instance, used a bi-gram vector space model for large text document classification, while [6] used word vector representation for short text classification. [7], however, proposed a text mining approach that operates by combining Convolutional Neural Networks (CNNs) and Support Vector Machine (SVM) with active SPL. Their classifier captures classes by progressively annotating unlabeled samples and verifies error-labeled sample data. Ensemble-based methods have also been evaluated for classification purposes. One recent work by [36] presents a fuzzy integral-based bug classification approach with ensemble and multi-RSMOTE data reduction techniques. Combining feature and instance selection was found to reduce the bug report dimensionality. This work addresses two major issues with bug reports, namely the high dimensionality of a bug report and problems

associated with word dimensionality. The approach is evaluated on Eclipse, GNOME, and Mozilla with standard evaluation metrics.

All these approaches are either text-dependent or (in the case of deep learning approaches) utilize a classifier trained for one dataset that may perform differently on another. Hence, the goal is to achieve stable and consistent results for datasets of a diverse nature. The solution to this problem is presented by our previous work on SPL-based bug labeling and classification in the paper by Mohsin and Shi [4]. This work addressed the issue of classifier performance instability on various datasets and achieved more than 94% precision on datasets of varying nature and size. Experiments were performed on the Eclipse, Mentis, and Redmine datasets.

To summarize, bug labeling is a basis of better triage and localization, and often underresearched. However, we suggest that better bug report classification can lead to better triage. Textual similarity-based approaches are quite common in bug report classification; however, since bug reports differ from natural language documents due to the challenging technical terminology they contain, the performance of well-known text classifiers decreases when used on bug reports. Therefore, other factors (such as concepts, content, feature selection, etc.) play an important role in correct classification.

3.3. Knowledge graphs and embedding

A comprehensive survey of knowledge graphs is presented by Ji et al. [28]. It presents the methods used to acquire knowledge (entities, relationship extraction), represent knowledge (embeddings), and perform temporal learning and graph completion. Similarly, a survey on knowledge graph-based recommender systems is also presented by Qin et al. [24]. The survey presents various kinds of knowledge graphs and their applications in different recommender systems. The authors presented different dataset scenarios and recommended the use of knowledge graphs in text-based learning. Major studies included were grouped into three main categories: (1) embedding-based methods, which involve entity/relationship learning and user preference recommendations; (2) Connection-based methods, which focus on the connection patterns among entities; (3) propagation-based methods, which include semantic relationships in a knowledge graph.

SBRNet leverages (1) and (2), i.e. knowledge acquisition and representation. However, it differs from these methods not only in terms of application and data but also in how it represents an overall combined picture of connections and entities/relationships that is further utilized for entity usage recommendation for triage purposes. Finally, this selected information is embedded in the proposed solution.

Another thorough survey of knowledge graph embedding within and outside of the knowledge graph domain is presented by Wang et al. [23]. Our work relates to the downstream task of entity classification. However, it again differs in the context and domain of application. We employ token association embedding from the bug entity text and connect it to the developer entity. Similarly, [27] proposed 2D convolution embedding, ConvE, for knowledge graphs. However, this work aims at predicting the missing relationships between entities in a graph; hence, its focus is on link prediction, making it different from our approach in terms of perspective. In [25], knowledge graph Attention Network (KGAT) is proposed, which models higher-order connections by also considering the related attribute information. The model additionally exploits the relationships among the instances to improve recommendation when evaluated on three datasets: Amazon-book, Last-FM, and Yelp 2018. Different from node embedding, community embedding is the distribution over the same node embedding space. This concept of community embedding, ComE+, is presented by Cavallari et al. [31]. It relates to SBRNet in the sense that we are also interested in embedding a certain set of information from the nodes and that the classified node vectors need not necessarily be of the same shape. Sui et al. [37], presented flow2vec for code summarization and classification through embedding. The evaluation results on 32 open source software showed 20% improvement for code classifier and 18% for code summarization when compared with the state-of-the-art methods. However, code has a structural set of keywords, whereas bug reports are free form text, thus processing of two artifacts is quite different. ANDRE, a malware clustering approach presented by Zhang et al. [38] clusters the weakly labeled malware using meta learning. The three layered deep neural network model effectively clusters the weakly labeled malware. Derbin and VIRUSSHARE datasets are used for evaluation.

All these studies reported above relate in some ways to our work; as discussed, however, the application, process and perspective is entirely different. Hence, SBRNet is a novel representation of bug report-based networks and utilization of node information, while its use of association embedding with SPL further makes it a distinct approach.

4. The proposed model

The proposed model is based on SBRNet for feature identification and the node information embedding. As outlined in the above definition and explanation of SBRNet presented in Section 2 (Figs. 1 and 2), the model operates on the attribute information related to the developer node $N4$ and class extraction based on the descriptive field d_r (summary) and label L in node $N3$. The model can be divided into two stages. In the first, it classifies the bugs into their corresponding types by referring to the summary of a bug report; in the second, it assigns the relevant developer to the corresponding bug for fixing, using categories assigned in the previous stage and information about the developer. An overview of the proposed model is presented in Fig. 3. The following subsection provides a detailed discussion of each step.

4.1. Problem formulation and model framework

Let $X = R^n$ be an n-dimensional bug feature space in SBRNet and $C = \{c_1, \ldots, c_n\}$ be a set of possible classes; moreover, let $D = \{D_1, \ldots, D_n\}$ be the set of developers forming node $N4$ in SBRNet. Consider a training set $\tau_s = \{(b_i, c_i)\}$, $i^n_{=1}$, where $b_i = \{b_{1i}, \ldots, b_{in}\} \in X$ is the ith bug's feature and $c_i = \{c_{i1}, \ldots, c_{in}\} \in C$ is the label vector of node $N3$ associated with b_i of node $N1$, obtained via association augmentation in $N2$. The goal of the bug classifier is to learn c_i from X to predict unknown bugs. Similarly, when assigning bugs to the developer, most bug classifiers attempt to map a bug b_i to a class c_i based on a certain classifier function δ, as follows:

$$\forall \delta({}^n_1 b_i) \leftarrow C_n \tag{1}$$

Here, δ is typically calculated as the weight of tokens needed to determine the textual similarity of words in the class and the inter-class difference.

For the proposed classifier function δ of our model, let us define $T = \{t_i, \ldots, t_k\}$ as the set of tokens ($N2$ of SBRNet) for each c_i ($N3$ of SBRNet); moreover, let $W = \{w_{1i}, \ldots, w_{nk}\}$ be the weight, while $F = \{f_i, \ldots, f_k\}$ denotes the frequency vectors for every t_i. Furthermore, let $z = \eta\ (C_i)$; t_i be the association identification function that yields the associations $t_{ci} \in \{t_{c1}, \ldots, t_{cn}\}$, generated through an A priori association algorithm for the candidate token C_t in each class C_i, which then prunes the non-frequent candidates C_{nf} based on the support function η to

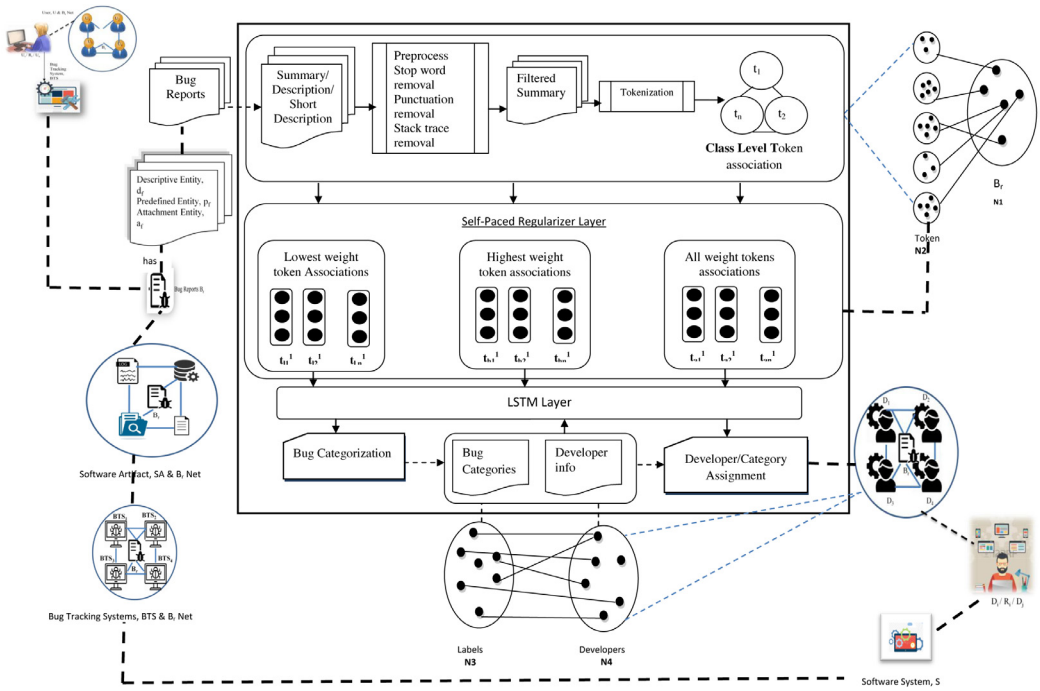

Fig. 3. The proposed model.

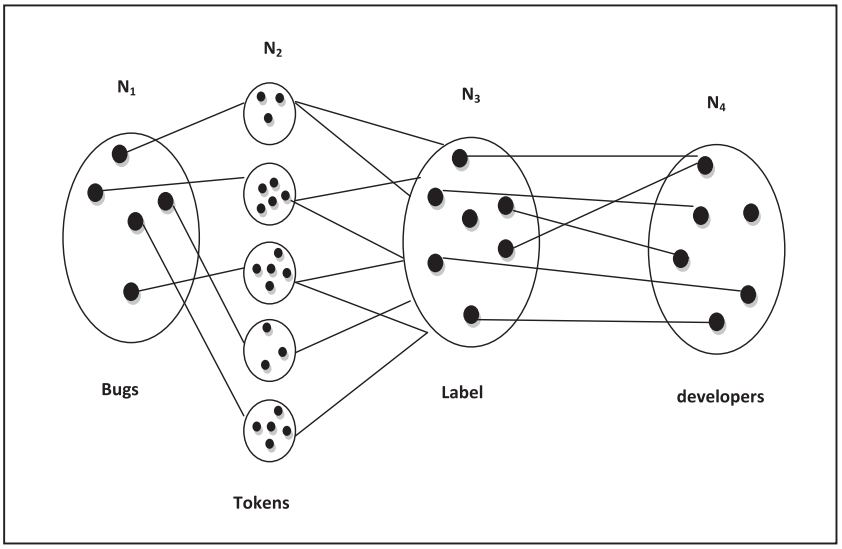

Fig. 4. Bug classification and assignment network.

generate the final set of itemsets given by the equation below (at a particular support threshold k, where $2 \leq k \leq 6$):

$$t_{ci} = C_t - C_{nf} \tag{2}$$

The input to the **S**elf-**P**aced **A**ssociation augmentation and **N**ode embedding (**SPAN**) is regularized by first inserting the data samples in a controlled manner, handled by the self-paced regularizer. The general optimization problem of SPL [39] is given by the equation below:

$$\sum_{i=1}^{n} v_i L(w, x_i, y_i) + f(V; \gamma); \tag{3}$$

Here, $x_i, y_i \in D$ and $L(w, x_i, y_i)$ represents the loss function while v_i is used for calculating importance, and $f(V; \gamma)$ denotes the self-paced regularizer function. For the problem under consideration, we used a modified definition of the function proposed in [4]. The regularizer function will operate in three steps, controlled by the parameter μ, as given by the below equation:

$$\delta_i = max_{0 \leq x \leq n} \sum_{i=0}^{k-1} w_{i+1}^{t} log \frac{n}{f_{i+1}^{t}} \tag{4}$$

The model operates in a self-paced fashion. In more detail, it first takes each association t_i^m that has the highest or the lowest weight tokens w.r.t. k (i.e., the frequent associations with the highest or lowest weight token in each class c_i). Subsequently, it embeds the remaining associations t_i^n for each class into the LSTM layer for further classification. Hence, the following equation in μ

yields the input embeddings for each class c_i:

$$\mu\forall({}_1^m t_i) + \mu\forall({}_{m+1}{}^s)t_i + \delta\forall({}_s^n)t_i \leftarrow C_k \quad (5)$$

Let $[w_1, w_2, \ldots, w_n]$ denote the association embedding of variable length l. At each time stamp τ, we concatenate the category and association vector at the last hidden state $[t_c; h_{\tau-1}]$. These embeddings are further passed through SPAN layers to produce the final classification using the softmax activation function and the Adam optimizer.

Finally, to assign bugs to the relevant developer D, the categorized labels b_l and developers D_i, are presented in Fig. 4 as N3 and N4; i.e., the developer and label information are embedded through the LSTM layer to yield the final bug assignment to the developer. The details of each step are further discussed below in the following subsections.

4.2. Proposed algorithm

A step-by-step explanation of the bug classification and assignment performed by the proposed algorithm is provided below.

Algorithm 1 SPAN-C: Self-Paced Classifier

1: Input $N_1 = (\tau_s)$, $N_2 = \{t_i, c_i\}$, $W = \{w_{1i},\ldots,w_{nk}\}$ and $F = \{f_i,\ldots f_n\}$, $t_{ci} \in \{t_{c1},\ldots,t_{cn}\}$
2: Output $N_3 = (b_i, l_i)$, C_k, class labels
3: (N_2, W, F) Preprocess(N_1)
4: **for** b_i in N_1 **do**
5: Remove stopwords, stack trace, punctuation from b_i
6: Apply stemming on b_i
7: **end for**
8: **for** filtered b_i **do**
9: **while** $N_1(b_i, c_i) \neq \emptyset$ and N_1 has classes C_i **do**
10: Form training classes C contacting each $b_i \in C$
11: **end while**
12: **end for**
13: **for** c_i in C **do**
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: **end for**
18: Return $(N2, W, F)$ // with f_i and w_i for each t_i having bug Id d_i
19: (t_{ci}) AssociationExtraction(T,W,f)
20: **for** (t_i, c_i) in N_2 **do**
21: get each token t_i in c_i
22: calculate $z = \eta\,(C_i)$; t_i
23: **while** T has tokens **do**
24: Generate candidate token-set
25: $t_c = \{1 - tokenset, 2 - tokenset \ldots n - tokenset\} \subset \{t_1, t - 2, t - 3 \ldots t_n\}$
26: **for** each $t_{ci} \in t_c$ **do**
27: prune non-frequent candidate token-set
28: Compute $support_{(t_i)} = \frac{l_{i \leftarrow t_i}}{n}$
29: $T_c \neq support$ are pruned i.e. neglected
30: **end for**
31: Compute frequent itemsets as
32: $t_{ci} = C_t - C_{nf}$ // where C_{nf} are non-frequent
33: **end while**
34: **end for**
35: Return t_{ci}
36: $(l_i)SPANClassification(t_{ci})$
37: **for** $\mu = 1$, $\mu\forall({}_1^m t_{ci})$ **do**
38: **if** $t_{ci} \leftarrow \delta_i = max_{0\leq x\leq n} \sum_{i=0}^{k-1} w_{i+1}^t log\,\frac{n}{f_{i+1}^t}$ **then**
39: LSTM layer $\leftarrow [V_{max}, S, \epsilon]$;
40: **end if**
41: **end for**
42: **for** $\mu = 2$, $\mu\forall({}_m + 1^s t_{ci})$ **do**
43: **if** $t_{ci} \leftarrow \delta_i = min_{0\leq x\leq n} \sum_{i=0}^{k-1} w_{i+1}^t log\,\frac{n}{f_{i+1}^t}$ **then**
44: LSTM layer $\leftarrow [V_{max}, S, \epsilon]$;
45: **end if**
46: **end for**
47: **for** $\mu = 3$, $\mu\forall({}_s^n t_{ci})$ **do**
48: **if** $t_{ci} \leftarrow min(\delta_i) < \delta_i < max(\delta_i)$ **then**
49: LSTM layer $\leftarrow [V_{max}, S, \epsilon]$;
50: **end if**
51: **end for**
52: Return l_i for each b_i

Algorithm 2 SPAN-A: Bug-Developer Assignment

53: Input $N_3, N_4 = (l_i, D_i)$
54: Output Mapping $b_i \leftarrow D_i$
55: **for** each $\hat{y} \in l_i$ **do**
56: **for** $D_i \in D$ **do**
57: get (A_i)AssociationExtraction$(\hat{y}, D_i)$
58: **if** support = threshold **then**
59: LSTM Layer $\leftarrow N = \{V, A\} \subset D$;
60: Map $V \leftarrow$ D through $max_f \sum log(P(N(u)|f(u)))$;
61: **end if**
62: **end for**
63: **end for**
64: Return $b_i \leftarrow D_i$

4.2.1. Data processing and class extraction

Every natural language problem must undergo several common steps (data cleaning, stop word removal, and stemming) to yields a reduced dataset that is fit for further processing. In our case, the bugs in node $N1$ are cleaned of punctuation, links, special characters, etc. as the first preprocessing sub-step. Subsequently, stop words are also removed (since these words do not support any particular learning process and are not meaningful). Finally, stemming is applied to reduce each word to its stem. This considerably reduces the feature set, and helps in finding frequent associations. Hence, for every bug b_i, in the dataset $X = (b_i, c_i)$, the pre-processing function (N_2, W, F) Preprocess(N_1) is applied as shown; this yields a filtered dataset S_f, as shown in Table 4. The steps of class extraction and preprocessing are outlined in Algorithm 1 (Lines # 1 to # 18).

After preprocessing, we obtain a filtered summary S_f for each bug b_i in $N1$. For the training model, we need the bugs from every class C_k. Each bug $b_i \in C = \{c_i, \ldots, c_k\}$ found in similar classes is collected to form the training class $C = \{c_i, \ldots, c_k\}$. We then need to tokenize each $b_i \in \{b_{i1}, \ldots, b_{in}\}$ into a space-delimited set of tokens $T = \{t_1, \ldots, t_k\}$ that forms node $N2$ of SBRNet; this enables us to create a vector space matrix for frequency and weight calculations.

4.2.2. Association extraction and self-paced data induction

The filtered data for each class obtained through the $N1$ and $N2$ nodes in SBRNet is then examined to determine the associations between each token and the bug type class. Let $I = \{I_1, \ldots, I_n\}$ denote the set of tokens in a particular class C_i. Each bug b_i contains a subset of I. Learning the association pattern

Table 4
Reduced token set of Eclipse bugs after applying the preprocessing steps (stopword removal, stack trace removal, stemming, etc.).

Before preprocess	After preprocess
[sfs] fix javadoc for options parameter	sf fix javadoc option paramet
[sfs] Eliminate issues discovered by FindBugs	sf elimin issu discov findbug
[sfs] Externalize names of SFS extension points to allow translation	sf extern name sf extens point allow translat
[sfs] Extension point schema files should be added to binary build	sf extens point schema file ad binari build
[sfs] Exception when running SfsExamplesTestSuite	sf except run sfsexamplestestsuit

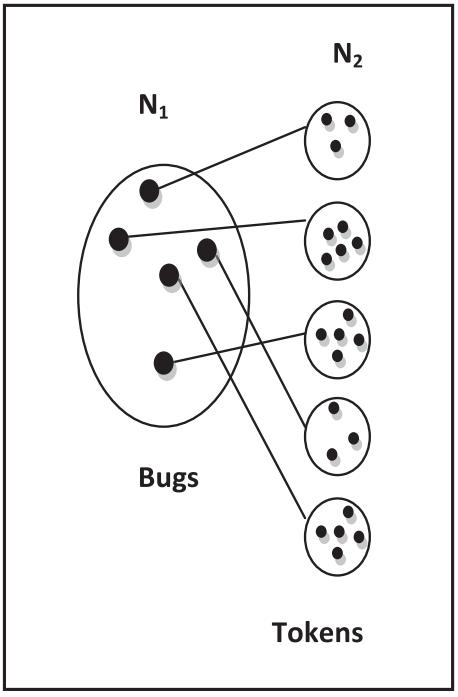

Fig. 5. Token association network.

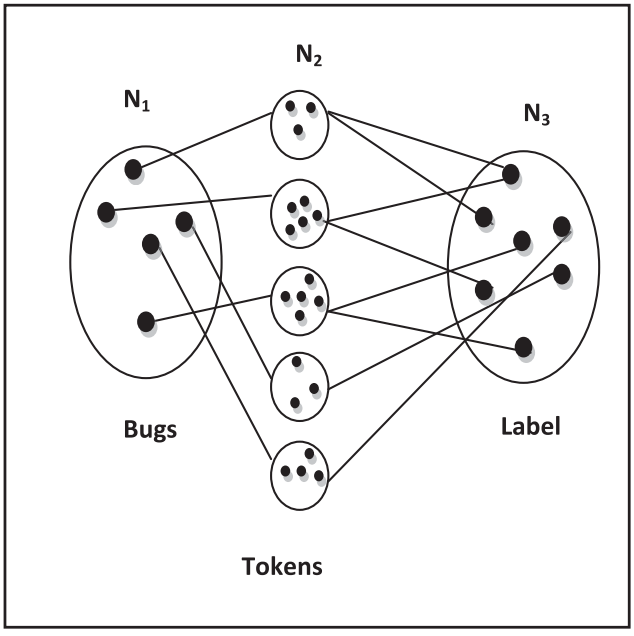

Fig. 6. Bug classification network.

requires finding the relationships between bug tokens in the corpus. In the first phase, $n+1$ itemsets are generated for each class; here, each token is taken as a 1-token candidate. Subsequently, the algorithm counts the occurrences; only those tokens that satisfy the given threshold (minimum support) k are put forth to the next iteration, while the others are pruned as described by Eq. (2). The minimum support k is calculated by the equation below. The token association network is depicted in Fig. 5.

$$support_{(t_i)} = \frac{I_{i \leftarrow t_i}}{n} \tag{6}$$

i.e., the number of itemsets containing the ith token from the complete set of tokens. This candidate generation and pruning process continues until there are no more frequent items to process. The function (t_{ci}) AssociationExtraction(T, W, f) in Algorithm 1 (Line # 19 to Line #35) explains the procedure step by step.

After the frequent itemsets are obtained for each class of bug, they are prepared for induction to the next step through frequency and weight calculation given by the function δ in Eq. (4). Let f_i denote the frequency of the ith token of b_i in class c_i. The frequency and weight calculations are given by the following equations:

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

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.

Given that the frequent patterns t_{im} having the highest weight W_m, tokens are created as one set of input that will be classified first. Similarly, the patterns with the lowest weight W_l tokens will be induced subsequently for categorization. Finally, all other associations will be put into the LSTM layer for classification (Algorithm 1; Lines # 37 to 51).

4.2.3. LSTM classification

The token associations obtained for each class through $N1$ and $N2$ are processed through **L**ong **S**hort **T**erm **M**emory (**LSTM**) for category classification for the node $N3$ (Fig. 6). Let $V = v_1, v_2, \ldots, v_{|V|}$ denotes a set of frequent itemsets that make up the item dictionary. Let $\epsilon = x_1, x_2, \ldots, x_{|V|}$ represent the embedding vectors with respect to the item dictionary V. Let $S = [s_1, s_2, \ldots, s_N]$, denote a sequence containing association tokens t_i for bug b_i at a certain time-step i, where l represents the length of the input sentence. The proposed SPAN model learns an embedding x_i for each item i in V. The goal is to predict the appropriate category label l_i for a given V by training the model association; the data is already in one-hot coded form as a result of the previous stages described in the above subsections. The steps are explained by the (l_i)*SPANClassification*(t_{ci}) in Algorithm 1 (Lines # 36 to 52). For every time-step t, the hidden state h_t and

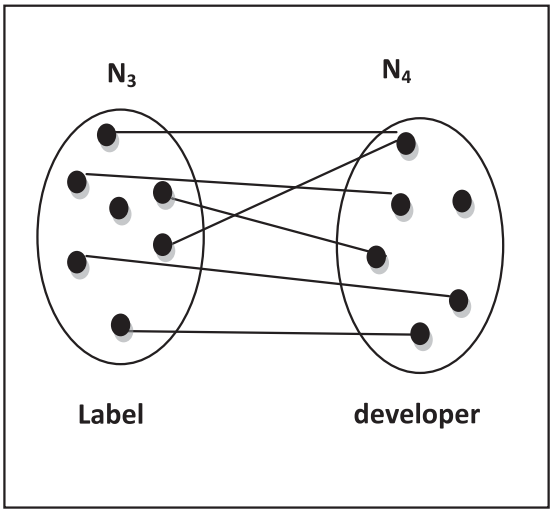

Fig. 7. Bug assignment network.

the memory cell m_t state are updated. The following Equations give the common function of LSTM.

$$\begin{bmatrix} In_o \\ Fo_t \\ out_t \\ \hat{C}_t \end{bmatrix} = \begin{bmatrix} S \\ S_A \\ l \\ \gamma \end{bmatrix} W.\begin{bmatrix} h_{t-1}, i_t \end{bmatrix} \tag{9}$$

$$m_t = Fo_t \bigodot m_{t-1} + In_t \bigodot \hat{C} \tag{10}$$

$$h_t = Out_t \bigodot \gamma_{mt} \tag{11}$$

In the above equations, i_t represents the input, while $\hat{C}$ denotes the present state of the single memory cell. Fo_t, and Out_t represent the forget gate activation function result and the output gate activation function result respectively. In addition, S_A represents the common softmax function (used widely for activation function multi-class classification), while γ represents the *tanh* function.

$$\hat{y} = softmax(V) \tag{12}$$

Here, $\hat{y}$ denotes the output vector of the model, which represents a probability distribution over the items v_i; each element $\hat{y}_i \in \hat{y}$ denotes the probability of the event that item v_i is associated with b_i. At any given time-stamp t, the loss function is defined as the categorical cross-entropy for multi-class text classification, as follows:

$$L(\hat{y}) = \frac{e^{s_i}}{\sum_j^C} CE = -\sum_i^C t_i log(L(\hat{y})_i) \tag{13}$$

where y denotes the one-hot coded labels. The Adam optimizer is then used to optimize the loss.

4.2.4. Bug assignment

Rather than following the current prevalent trends in for bug-developer assignment, we make use of the categories obtained through SPAN to further proceed towards mapping the appropriate developer. The details for LSTM are similar to those discussed in Section 3.3. By adopting this approach, we avoid the need for extensive feature management, which is otherwise required when summary, developer, and a large number of other data dimensions are explored. This not only reduces the time required to assign bugs, but also improves the accuracy, as will be discussed in the results section.

For every bug b_i, there exists a developer D_i who is responsible for fixing that bug. Given a set of developers, $D = \{D_1, D_2, \ldots, D_n\}$ a developer D_i is to be assigned to resolve b_i. The bug labels $\hat{y}$ thus obtained in the last phase are mapped to D_i by passing them again through the LSTM layer. This time, we simply input the nodes N_3 and N_4; i.e., the labels $\hat{y}$ and developers d are input as text sequences at the given time-stamp t along with h_t hidden states, while a *softmax* activation function is applied for every b_i to produce the desired classification result. Let $N = (V, A)$, $V \subset D$ be the node set, while A denotes the associations among the D and $\hat{y}$ sets. A set of node instances N can be associated through a mapping function $f : V \leftarrow D$ from labels to developers. A node is mapped to a category $\hat{y} \in \{y_1, y_2, \ldots, y_n\}$, if there exists an association $A_i \in N$ that satisfies the support threshold. The following equation defines the probability of the two nodes being mapped together:

$$max_f \sum log(P(N(u)|f(u))) \tag{14}$$

As the bug labels are already classified, the neighborhood for each node mapping is reduced to direct neighbors and insight feature inspection is not required. A D_i is mapped to label l_i using the above function. Algorithm 2 (Lines # 53 to 64) outlines the major steps involved in the bug-developer assignment procedure. The weightage and frequency labeling for each token/keyword is already performed at the classification stage; hence, the only work remaining is to map that bug to its corresponding developer. In instances of one-many or many-many class-developer mapping (e.g. A, B, and C are three developers that are making changes to a UI class bug, or UI and CSS bugs are handled by multiple developers, i.e. if multiple bugs are assigned to one developer for fixing or many bugs are assigned to various developers.), the augmented associations from the previous stage, the model learns based on frequency, weight and labels that a particular $b_i \in b$ containing one set of keywords in a particular class is to be mapped from a set of available $D_i \in \{D_1, D_2, \ldots, D_n\}$. The assignment process is simply to map the two nodes $N3$ and $N4$ based on the knowledge embedding from $N1$ and $N3$ (Fig. 7). Furthermore, the data that is curated for the model contains bugs and the developers who resolved these bugs. The more bugs fixed by the developer of a particular class, the more experienced he is; thus, expertise is considered automatically during association augmentation, in that developer A is associated with a higher number of labels $\hat{y}$ when compared to developer B. Hence, it is most likely that the model will assign the highest weight to developer A while passing through the LSTM layer. However, as noted above, we aim to remove the additional fields for bug assignment in order to simultaneously increase simplicity and efficiency; therefore, for bug-tokens-developer associations, the expertise (age, years of experience) is not considered directly.

5. Experiments and results

This section first lists the research questions devised for the purpose of evaluating our model, followed by details of the dataset selection and preprocessing. Subsequently, the evaluation metrics are presented. Finally, the findings obtained through the series of experiments are discussed in detail. Python (Spyder/Anaconda) is used as the IDE with the Python development environment.

5.1. Research questions

The research questions devised to evaluate the model are as follows:

- RQ1: Does SPAN exhibit less variance in performance than other approaches when tested on multiple datasets?

- RQ2: Does the proposed approach outperform other approaches?
- RQ3: Is the model robust with different sizes of training samples?
- RQ4: Does SPL help to improve the performance of the proposed classifier? 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 (Sections 5.4.4 & 5.5.3; Figs. 13 & 18; Tables 7 & 9). The third research question, RQ3, is formulated to investigate the accuracy variation and robustness of the model when applied to datasets and training samples of various sizes (Sections 5.4.1, 5.4.2, 5.5.2 & 5.5.3). Finally, RQ4 evaluates the effect of SPL on improving the classifier performance (Section 5.4.3).

5.2. Datasets and preprocessing

The proposed approach has been implemented and evaluated on the following datasets: Bugzilla (using the data for the Eclipse Project[1] while adding keywords to search bugs from 2010 to 2020), Redmine[2] (containing bugs from 2008 to 2020), Mentis[3] (containing bugs from 2004 to November 2020), the Git-Hub[4] dataset for the Eclipse project (containing 168,000 reported bugs from the Eclipse project until 2013), and 10,000 bugs tagged in the Eclipse dataset.[5]

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

We merge all datasets rather than address bugs from different projects separately. This choice was made for the following reasons: (1) The project-specific bugs contain similar kind of token-sets (words), meaning that learning on project-specific bugs can introduce bias. To prove our claim that SPAN provides substantial results for both textually dependent and independent bugs, we gathered bugs from diverse projects, platforms, sizes, keywords, and bug tracking systems and put them all together in a unified dataset to remove any bias. (2) We needed labeled bugs to train our model and eventually to evaluate the difference between actual and model labeling. The datasets available for every individual project contain a far smaller number of labeled bugs. For instance, out of more than one million bugs collected, we had only 30,000 bugs that were already labeled. Therefore, to ensure our model could be tested on the largest dataset possible (as deep learning models trained on larger datasets tend to be more effective), we created a unified collection of data from various projects. (3) Eclipse bug reports are used for classification and triage by taking into account the bug report summary, description, and other fields since 2004 by Cubranic and Murphy [8] who used the Eclipse bug reports for triage while dealing with triage as a classification problem. They used the dataset in the same manner as we do and selected a total of 15,859 Eclipse bug reports. Xuan et al. [9] evaluated their semi-supervised bug triage approach with Bugzilla reports starting by taking IDs from 150001 to 170000 as an entire dataset rather than being project-specific. This was followed by [4,5] utilizing Eclipse bug report information without segregating it into different projects.

For very large datasets such as Eclipse, we have a huge number of these tags/labels that 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. Moreover, for Redmine and Mentis, the bugs were already tagged in the available dataset under the "category" field, containing 52 and 54 bug categories, respectively. Therefore, these datasets can be used without the need for 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 metrics.

The rationale for choosing Redmine, Mentis, Bugzilla, and Eclipse was based on our aim to retrieve data from every aspect; for instance, while the Redmine dataset is small, the summary contains very diverse words used to define the bug category. Similarly, the Eclipse and Bugzilla datasets provide data from different projects and platforms (e.g. Windows, Linux, etc.). Moreover, the bug category is tagged inside the short description and mentioned implicitly. Therefore, we created one unified dataset so that our model could be validated on every type and size of dataset. The individual characteristics of each dataset are presented in Table 5, which represents the total number of records extracted (size), number of records that has unique class labels (labeled data), number of tokens each bug has on average, and learning complexity (i.e. how much textual similarity can be obtained between the words contained in the bug and its category label). The higher the proportion of similar features, the easier the sample.

Bugs from multiple bug tracking systems though have unique characteristics, however, we are concerned with the summary or short description, bug type, and developer assigned. Therefore, after retrieval, we extracted the bug_ID, short description/summary, bug category, and assigned developer from each dataset. For merging (1) Bug IDs are generated collectively by a random number generator once all data is gathered. (2) Bug description is preprocessed and stemmed to obtain a cleaned dataset free of stopwords, link traces, etc. (3) Bug category is to be processed for Eclipse/Bugzilla based datasets as bug category or type is not mentioned explicitly rather tagged in square brackets at the start of the description, therefore, we applied preprocessing steps and put the tags in a separate column of bug type for compatibility with other datasets like Mentis and Redmine. Subject, category, etc. column captions are changed to one column name i.e. bug_type. (4) Bug fixers also come under various headings like developer, assignee, etc. we put them all together in the bug_fixer column. (5) The distribution bugs in each type of dataset are different initially but after combining we group similar classes of bugs from every dataset. Eventually, our model works on the uniform distribution of each bug class.

5.2.1. Baselines and settings

We ran our experiments on a tested environment running a 64-bit Windows and Intel Pentium Core i3 3.4-GHz CPU and 32-GB RAM. We used Tensorflow to conduct the experiments. The maximum sequence length is set as 20, embedding dimensions = 100, and the batch size is set as 32 with 10 training epochs.

1 https://bugs.eclipse.org/bugs/.
2 https://www.redmine.org/issues.
3 https://www.mantisbt.org/bugs/view_all_bug_page.php.
4 https://github.com/ansymo/msr2013-bug_dataset.
5 https://www.kaggle.com/monika11/bug-triagingbug-assignment/data.

Table 5
Characteristics of datasets: Three datasets are shown here. Since the GitHub dataset also contains 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

The dropout rate is set to 0.2. LSTM and Dense layers are applied. Categorical cross-entropy and the Adam optimizer are used as the loss function.

Bug classification is not done as a separate task in most existing research. The approaches chosen for comparison are therefore those that are only somewhat related to ours. In the literature, the approaches used specifically for dealing with the problem of classification in bug reports using summary or description were variants of Naive Bayes [40]. Other approaches selected for comparison, were either widely used bug prediction techniques or text classification techniques (both baseline and state-of-the-art methods) of varied nature to cover the comparisons from every aspect. The list of baselines for comparison are as under:

- **Bayesnet**: Bayesian network is a probabilistic model which provides a relationship between nodes and edges in a graph representation. It is a classic algorithm for classifying and identifying relationships, therefore, we included this approach to compare our model.
- **Logitboost**: It is a boosting algorithm based on additive logistic regression, which is widely used as a solution to classification problems. Hence, this is also included in our comparisons.
- **J48**: C4.5 (J48) is a well-known classification algorithm and is commonly used for classification problems. Thus, it is also selected for comparison.
- **SMO**: We opted Sequential Minimal Optimization (SMO) algorithm to be compared with our method that has been used widely for classification-related problems.
- **Random Forest & Treebag**: To include ensemble methods in our comparison we chose random forest and treebag, a combination of multiple decision trees.
- **CNN & MLP**: Neural Networks (NNs) have also been recently introduced into the field of bug classification. Although these are mostly used for identifying bugs from source code repositories, we have conducted a comparison with [7], in which the authors used CNNs and Active SPL to perform unlabeled text classification.
- **KNN**: The work of Zhang et al. [16] is also compared. These authors used KNNs, based on instance similarity between a new bug report and similar reports from the historical repositories.
- **RFSH & RFSTM**: We compared our bug assignment model with the latest approach presented by [22], where authors used RFSH (**R**egular **F**eatures, **S**ummary, **H**ybrid) and RFSTM (**R**egular **F**eatures, **S**ummary, **T**opic **M**odeling) to assign bugs based on developer expertise.

To conduct our comparisons, we used our dataset (containing a majority of bugs from Eclipse/Bugzilla) in a Python environment. The first seven (except for KNN) baseline methods in Table 7, for classification, are executed using the Weka software (as these are the well known implementations and publicly available) whereas the new methods (KNN, CNN, and MLP) are implemented according to the approach defined in their respective papers. To facilitate fair comparison, we used the same dataset that is used for our model.

Rather than priority calculation, for bug assignment, we considered the developer classification results of the approach. Terms of similar bugs were collected in classes/clusters (as in our preprocessing step, with frequency, weight, and neighborhood association calculation). Feature augmentation of the original vector was then performed. Unlike the original approach, these vectors are then used to classify the bugs into types and then again to facilitate assignment to the relevant developer. Although the settings were modified, the results obtained were almost the same as those reported in [22], with a minute difference of a few decimal points; for instance, RFSTM's 0.902668 accuracy can be rounded off to of 0.903%, which is similar to [22]. To facilitate comparison with the CNN-based classifier, we used the same settings and tested on the Eclipse dataset obtained from Bugzilla, as those described in [22]; we also obtained similar results, since similar bug summaries were utilized. However, instead of priority classification, we obtained the developer classification being hot-encoded.

5.3. Metrics

In this section, the classification and the assignment of the model are evaluated with reference to several evaluation metrics. We calculate the accuracy, specificity, and robustness of the proposed model on a unified dataset that contains various aspects of the data. A comparison between our method and baseline methods is also presented in this section. In the below we define the metrics used to evaluate the performance and stability of our model.

True Predictions (TP) – Correctly predicted classes; for example, the original class was CSS and was also predicted as CSS.

False Predictions (FP) – Classes that are predicted incorrectly; for example, a CSS class is predicted as Compatibility.

Missed Predictions (MP) – Classes that are not identified or mapped to any of the classes in the dataset.

Accuracy – Ratio of correctly predicted observations to total observations.

$$Accuracy = (TP + FP)/(TP + FP + MP)$$

Precision – Ratio of correctly predicted observations to the total predicted observations.

$$Precision = TP/(TP + FP)$$

Recall (Sensitivity) – Ratio of correctly predicted positive observations to all observations in an actual class.

$$Recall = TP/(TP + MP)$$

F1 score – Weighted average of Precision and Recall.

$$F1 = 2 * (Recall * Precision)/(Recall + Precision)$$

Kappa – A metric that compares an Observed Accuracy with an Expected Accuracy (random chance) given by the Equation below

$$Kappa = \frac{observedAccuracy - expectedAccuracy}{1 - expectedAccuracy}$$

Hit Rate – The ratio of correct predictions to the total number of predictions, calculated by the below formula:

$$hitRate = \frac{\sum_c TP_c}{\sum_c TP_c + \sum_c FN_c}$$

where TP is true positives (i.e. correct predictions) and *FN* denotes false negatives in a confusion matrix.

Table 6
Performance on varied test/train ratios.

Test/Train ratio	Loss	Accuracy	Precision	Recall	F-1 score	MCC	Cohen's Kappa	MAP	ARR	Hit rate
0.90	2.30	0.50	0.77	0.41	0.53	0.50	0.50	0.53	0.52	0.52
0.80	1.56	0.69	0.83	0.63	0.71	0.68	0.68	0.70	0.69	0.69
0.70	1.20	0.77	0.87	0.73	0.80	0.76	0.76	0.77	0.77	0.77
0.60	0.84	0.84	0.90	0.82	0.86	0.84	0.84	0.84	0.84	0.84
0.50	0.62	0.89	0.93	0.87	0.90	0.89	0.89	0.89	0.89	0.89
0.40	0.37	0.93	0.96	0.93	0.94	0.93	0.93	0.94	0.94	0.94
0.30	0.25	0.96	0.97	0.95	0.96	0.96	0.96	0.96	0.96	0.96
0.20	0.19	0.97	0.98	0.96	0.97	0.97	0.97	0.97	0.97	0.97
0.10	0.13	0.98	0.99	0.98	0.98	0.98	0.98	0.98	0.98	0.98

MAP – Macro Average Precision (MAP) is the average of precision obtained for each class under different sets. It is defined as follows:

$$MAP = \frac{\sum_c TP_c}{\sum_c TP_c + \sum_c FP_c}$$

where TP denotes true positives for every class (i.e. correct predictions) and FP is false negatives in a confusion of each class matrix.

More simply it is the average precision of each class, defined as follows:

$$MAP = \frac{1}{N}\sum_{1}^{N} P_i$$

where, N is the total number of classes and P_i is the precision for the ith class.

ARR – Average Recall Rate (ARR) is the average recall obtained for each class under different sets. It is defined as follows:

$$ARR = \frac{1}{N}\sum_{1}^{N} R_i$$

where, N is the total number of classes and R_i is the recall for ith class.

These metrics are tested with 10-fold cross-validation and compared with the results obtained on the same testing and training sets, enabling evaluation of the model's stability, performance, and robustness.

Experiments assessing the classification and assignment of bug reports are conducted separately and reported under separate sections.

5.4. Classifier results

To evaluate our classification results, a series of experiments have been conducted to assess various aspects. Results so obtained are reported in the following sections.

5.4.1. Performance comparison with varied test/train ratio

To evaluate our classifier performance, we selected different proportions of testing and training datasets. Beginning from only 10% testing and 90% training sets, the proportion is gradually decreased with a step size of 10% each time until 10% training data and 90% test data remains. The evaluation metrics used are listed in Table 6. As the table shows, accuracy, precision and hit rate (i.e.the proportion of correct predictions relative to overall predictions) remain above 90% even when less than 50% of the training data is available. The highest values for accuracy and precision are 98% and 99% respectively; these drop gradually until the training data proportion is at 10%. However, the model differentiates well between the classes for the rest of the proportions. Recall, F1 score and other parameters exhibit similar behavior, even at 40% training data and 60% test data. In addition, the macro metrics (MAP, ARR, hit rate), computed for each class individually, also exhibit a similar trend; there is very little difference between micro and macro computations, as is evident from Table 6. Furthermore, there is only a slight difference in the formulas of Hit rate and ARR; specifically, the two values are almost equal under multi-class settings. For instance, values of hit rate and ARR (0.51613 and 0.51576), for 0.90% of test set both round off to 0.52. Table 6 suggests that the classifier performance under various parameters is quite substantial, especially given that the precision for bug type classification is significant and remains intact even for small training sets. The graphical presentation of the results in Table 6 (see Fig. 8) suggests that precision exhibits a much more stable and higher curve when compared to the other metrics. However, the rest of the curves also mimic precision at testing proportions of 40% and below. Beyond that testing proportion, these parameters begin to deviate from the precision curve, as less training data is involved and wrong classifications are also encountered alongwith with the correct and incorrect results. Thus, the correct predictions (hit rate) and other metrics show lower values when compared to precision. Similarly, Fig. 9 depicts the loss curve, which rises to 2.3 and then gradually settles down at 0.1 when testing data is only 10% of the whole set. It also remains below 1.0 for a test set ratio of 60% or below. Hence, overall, the classifier exhibits stable performance on smaller and varied sets of testing data.

5.4.2. Performance under 10-fold cross-validation

To further evaluate the model, we also included the 10-fold cross-validation results by keeping one set as the training set, assigning the rest to testing, and choosing a different testing dataset each time to assess the behavior of the classifier on each fold. Fig. 10 plots the accuracy and loss variations for the testing and training sets. Training and validation accuracy stabilizes at the fourth epoch, while the loss drops to near zero in the sixth epoch.

5.4.3. Effect of self-paced learning

To assess the effect of SPL on our classifier performance, we also gathered results without incorporating SPL into the model. The results so obtained are illustrated in Figs. 11 and 12 comparison of Fig. 8 and Fig. 11 demonstrate that without SPL, the results are poor and exhibit some fluctuations (e.g. at 0.70 and 0.50 of test ratio) in precision and accuracy. The highest value of precision attained without SPL is below 85%, or approximately 15% less when compared to the results in Fig. 8. Moreover, the loss is high and remains above 1.0 without SPL, while with SPL this value is nearly zero (Fig. 9). These results therefore validates the effect of SPL on the model, in that the classifier can be seen to stabilize if the data is induced in a controlled and effective manner.

5.4.4. Comparison with other approaches

In the literature, very few studies have addressed software bug classification in isolation, as this area is normally dealt with as a part of some bigger problem; therefore, there are only a few

Fig. 8. Performance variation with the varied ratio of testing and training sets. Observations began from 90% testing datasets and were gradually reduced by 10% each time. Lowest precision with only 10% training and 90% test is above 75%. Behavior of the model on various metrics is represented by each colored curve.

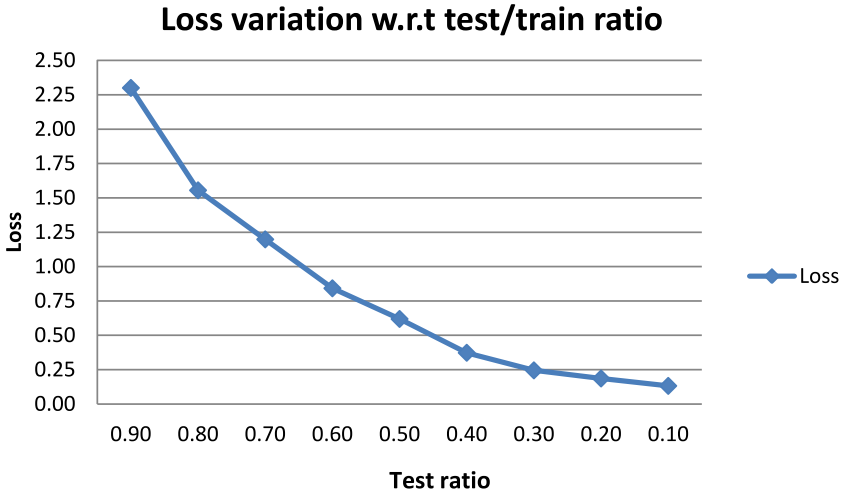

Fig. 9. Loss variation with test/train sets. Observations began from 90% testing datasets and gradually reduced by 10% each time. Loss reduces to 0.1 with 90% training and 10% testing.

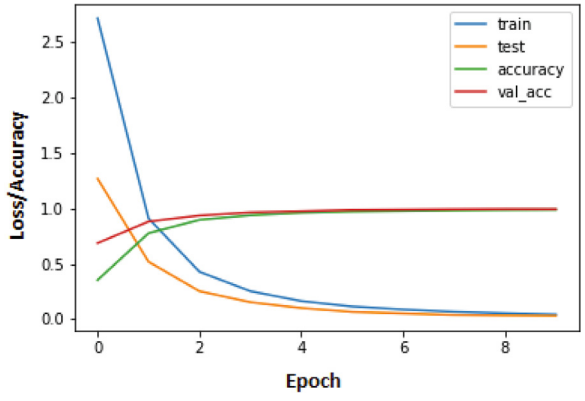

Fig. 10. Training and validation loss and accuracy. Validation accuracy becomes stable and approaches 1 after the fourth epoch and continues to exhibit the same behavior at the seventh of the epochs. Similarly, loss (blue & orange lines) settles to zero at the seventh epoch.

Table 7
Performance comparison with other methods when testing and training data is obtained from the same dataset.

Algorithm	Accuracy	Precision	Recall	F-score	kappa
Bayesnet	0.592	0.800	0.592	0.570	0.529
KNN (HengChang et al. {2016]	0.912	0.952	0.912	0.934	0.903
logitboost	0.611	0.890	0.611	0.200	0.551
J48	0.346	0.619	0.346	0.630	0.186
SMO	0.946	0.968	0.946	0.972	0.890
Random forest	0.950	0.972	0.950	0.984	0.900
TreeBag	0.955	0.990	0.966	0.976	0.962
MLP	0.953	0.967	0.952	0.959	0.950
CNN (TingYe Zheng et al. 2018)	0.932	0.947	0.932	0.939	0.930
SPBC (2020)	0.964	0.977	0.964	0.971	0.960
SPAN	0.994	0.998	0.989	0.984	0.981

methods available for comparison. Accordingly, we have selected well-known text classifiers and state-of-the-art bug classification approaches to compare with our model.

Table 7 and Fig. 13 depicts the performance variations among different classification methods. As is evident from the results of Table 7, our techniques outperform all other existing baseline approaches in terms of accuracy, precision, recall, and kappa.

Fig. 11. Performance at various test/train ratios without incorporating SPL. The highest precision obtained with 90% training is below 85%, which is quite low when compared to the results obtained by incorporating SPL (average 98% precision).

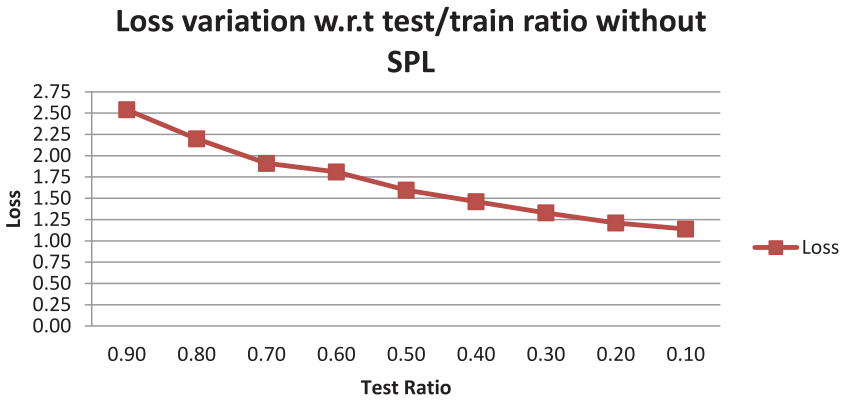

Fig. 12. Loss at various test/train ratios without incorporating SPL. Loss is reduced to below 1.25 at 90% precision; by contrast, this figure becomes zero when results are obtained by incorporating SPL.

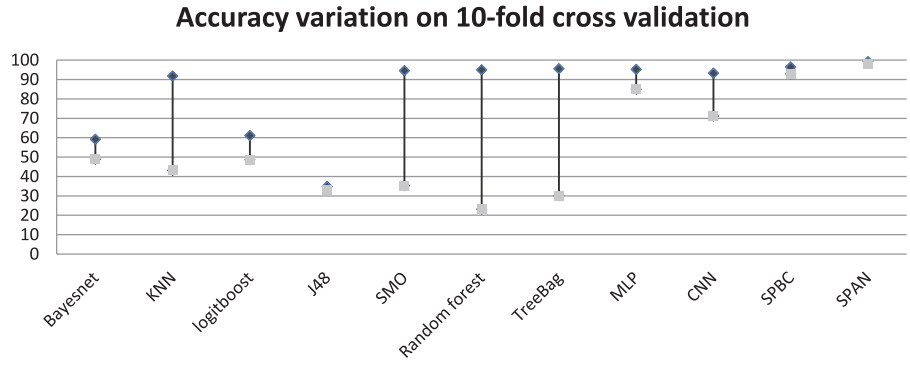

Fig. 13. Accuracy variation on 10-fold cross validation. The vertical black lines indicate indicates the highest and lowest accuracy values of comparison methods relative to SPAN. It is clear that SPAN not only achieves the highest accuracy, but is also more stable compared to other methods (i.e. exhibits less variation in results when tested on different data chunks).

Furthermore, when validated with 10-fold cross-validation, the results are even more convincing (as shown in Fig. 13). The candle-stick figure illustrates the variations of each method with 10-fold cross-validation on the same test/train sets. It is clear that our approach not only outperforms the others but is also the most stable on various types and proportions of testing datasets, while other approaches suffer from performance variations of 10% to 70% on average, SPAN is more stable than these comparison approaches while also achieving but also achieved higher performance.

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 calculations on the accuracy, precision, recall, and F-measure are conducted independently, where the unit of analysis is the dataset. Table 8 presents the ANOVA analysis results. As the table shows F $>$ Fcrit and p-value $<$ (alpha $=$ 0.05) are true for precision and accuracy. This suggests that the factor of interest (the use of different approaches) is associated with a significant difference in accuracy and precision.

Fig. 14. Bug assignment performance at various test/train ratios. Observations began from 90% testing data-sets and gradually reduced by 10% each time. The lowest precision (with only 10% training and 90% testing) is still above 75%. Behavior of the model on various metrics is represented by each colored curve.

Table 8
Single-factor ANOVA representing significance of comparison among various approaches.

SUMMARY					
Groups	Sum	Average	Variance		
Bayesnet	3.0832	0.61664	0.011164		
KNN (HengChang et al{2016]	4.613	0.9226	0.0004008		
logitboost	2.8629	0.57258	0.0607484		
J48	2.1272	0.42544	0.0372919		
SMO	4.722	0.9444	0.0010708		
Random forest	4.756	0.9512	0.0010332		
TreeBag	4.8486	0.96972	0.000184		
MLP	4.7814414	0.9562883	4.836E−05		
CNN (TingYe Zheng et.al 2018)	4.6804401	0.936088	5.02E−05		
SPBC (2020)	4.8359	0.96718	4.551E−05		
SPAN	4.946	0.9892	4.87E−05		
ANOVA					
Source of variation	SS	MS	F	P-value	F crit
Between groups	2.0067878	0.2006788	19.694425	3.937E−13	2.053901
Within groups	0.4483434	0.0101896			
Total	2.4551312				

Table 9
Performance comparison with other methods.

	RFSTM	RFSH	CNN	SPAN
Accuracy	0.9026	0.9408	0.9097	**0.9672**
Precision	0.8945	0.9248	0.9069	**0.9465**
Recall	0.9025	**0.9409**	0.8978	0.9243
F-score	0.8987	0.9326	0.8992	**0.9341**

5.5. Bug assignment results

Experiments are also conducted to evaluate the assignment of bugs to the relevant developer. Results are presented in the following subsection, with reference to the metrics defined in Section 5.3.

5.5.1. Performance comparison with varied test/train ratios

Fig. 14 presents the performance of the model when mapping a specific type of bug to the appropriate developer at various test/train ratios. When the training data size is gradually decreased with a step of 10% each time, the performance parameter also shows variations. However, the curve for each metric so obtained does not exhibits any abrupt changes and remains stable, with 95% precision at most. Precision remains above 85% with a test data proportion of 50%. All other metrics, including F-1 Score, recall, true prediction rate (i.e. hit rate), etc., begin to deviate from the precision curve after the higher initial values at (around 40%) and higher test ratios; the number of false and missed predictions gradually increases as the proportion of training data is reduced. Similarly, loss variations at particular test/train proportions are also plotted in Fig. 15, starting at 2.2 and reaching 0.2 with trained data at 90%.

5.5.2. Performance under 10-fold cross-validation

To assess the performance of the bug assignment model, we further performed experiments with 10-fold cross validation; here, one sample is kept as test data each time, while the remainder is used to train the model. Figs. 16 and 17 plots the accuracy and loss variation at different epochs. Accuracy ranges from 65% to 96%, while loss varies from 1.05 to 0.1. Test accuracy becomes stable and increases uniformly from the third epoch onwards.

5.5.3. Comparison with other approaches

Fig. 18 and Table 9 depicts the performance of the proposed model in terms of its assigning of bugs to the most appropriate developer using the latest triage techniques. The results reveal that our approach achieves better performance in terms of accuracy, precision and F-score. SPAN's performance remains around

Table 10
Single-factor ANOVA representing the significance of using the SPAN assignment model w.r.t. other approaches.

Anova: Single factor					
SUMMARY					
Groups	Sum		Average		Variance
RFSTM	3.6		0.9		1.46667E−05
RFSH	3.74		0.935		5.86667E−05
CNN	3.61		0.9025		9.16667E−05
SPAN	3.771		0.94275		0.00034225
ANOVA					
Source of variation	SS	MS	F	P-value	F crit
Between groups	0.005795187	0.001931729	15.2329555	0.000215106	3.490294821
Within groups	0.00152175	0.000126813			
Total	0.007316937				

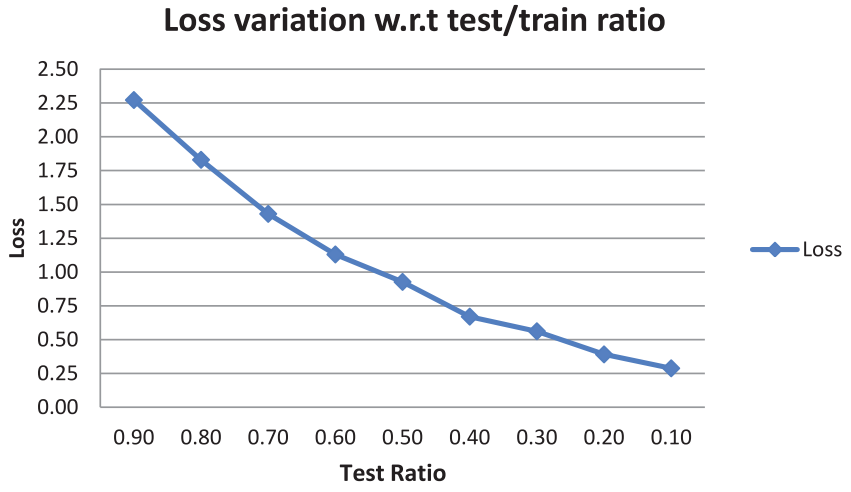

Fig. 15. Loss at various test/train ratios for bug assignment. Observations began from 90% testing datasets and gradually reduced by 10% each time. Loss reduces to around 0.25 with 90% training and 10% testing.

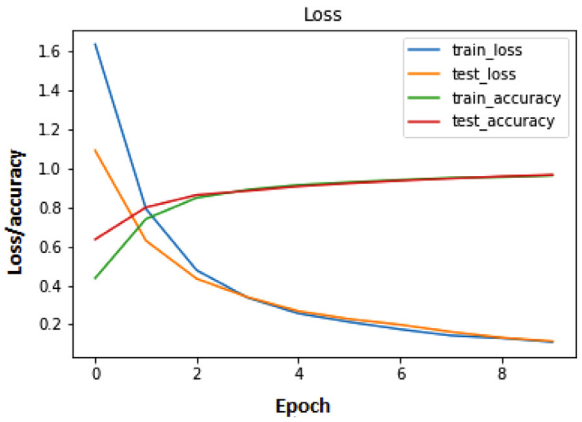

Fig. 16. Training and validation loss/accuracy for bug assignment. Validation accuracy becomes stable and approaches 1 after the third epoch, then continues to rise gradually for the remainder of the epochs. Similarly, loss (blue & orange lines) settles around zero at the ninth epoch.

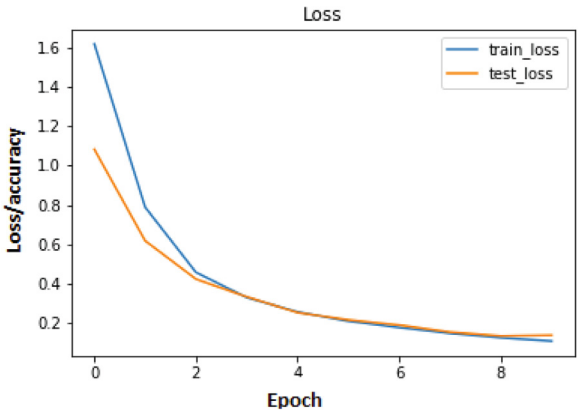

Fig. 17. Training and validation loss uniformly decreases from the third epoch and settles around zero at the eight epoch.

or above 94%, with accuracy and precision considerably higher when compared to other parameters. While the RFSH recall value is better than SPAN's the difference between them is only 0.017. ANOVA analysis in Table 10 represents the significance of using SPAN over other approaches.

Table 11 lists the results of a two-way ANOVA conducted to assess the differences between SPAN and SPBC. This ANOVA is carried out separately as there is minimal difference between the two approaches. Here, again, the F > Fcrit and p-value < (alpha = 0.05) results reveal that the difference between the two approaches is nevertheless significant. Furthermore, ANOVA for the association model reveals that the results are also significant (Table 11).

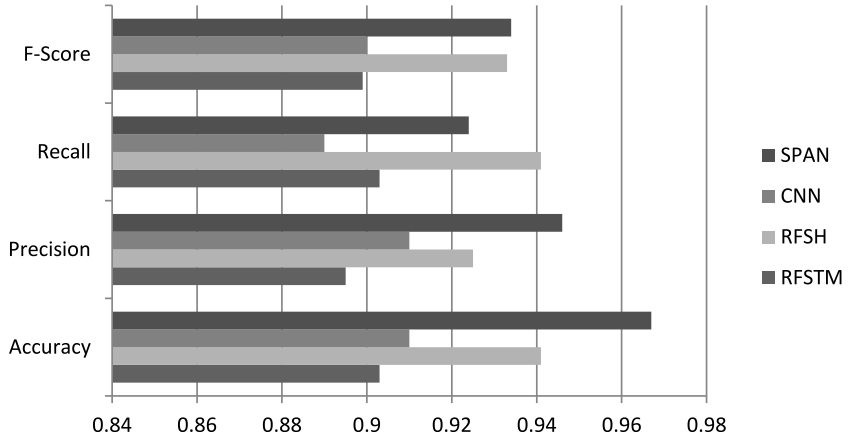

Fig. 18. Performance comparison with other approaches.

Table 11
Two-way ANOVA representing the significance of SPAN and SPBC.

Anova: Two-factor without replication			
SUMMARY	Sum	Average	Variance
	1.9803	0.495075	0.3010218892
	1.9609	0.490225	0.2949115492
	1.9379	0.484475	0.2939144092
	1.9477	0.486925	0.2907880892
SPBC	1.9283	0.482075	0.2847779558
	2.0078	0.50195	0.2609114767
	2.0358	0.50895	0.2351518767
	1.8506	0.46265	0.2613419033
	1.8242	0.45605	0.2536109433
	3.934	0.9835	1.76667E−05
	3.881	0.97025	7.625E−05
	3.844	0.961	8.46667E−05
	3.754	0.9385	0.000192333
SPAN	3.593	0.89825	0.000599583
	3.413	0.85325	0.001382917
	3.169	0.79225	0.00346825
	2.857	0.71425	0.006832917
	2.21	0.5525	0.023625
Accuracy	15.8798	0.882211111	0.014792873
Precision	16.7457	0.930316667	0.00340589
Recall	7.45	0.413888889	0.182213987
F-1	8.053	0.447388889	0.182994487

ANOVA					
Source of variation	SS	MS	F	P-value	F crit
Rows	3.08266607	0.181333298	2.692083405	0.003280769	1.827147
Columns	4.10287208	1.367624027	20.30381613	8.57807E−09	2.7862288
Error	3.43525695	0.067357979			
Total	10.6207951				

6. Threats to validity and limitations

The unified dataset was created by gathering bugs from multiple bug tracking repositories and datasets available online. However, the class distribution is not uniform; moreover, the difficulty levels for learning are diverse and the sizes are also varied. This may lead to a biased learning rate affected by the data distribution. This risk is mitigated by testing the model on various test/train proportions and utilizing random sampling; results revealed that the model is capable of maintaining good performance on all types of distributions and proportions.

Moreover the, model is tested on historical bug repositories, which might not be a truly representative of runtime bugs. To address this potential issue, first, 10-fold cross-validation with unseen data insertion is carried out. Results validate that the model is capable of identifying various bug types with improved accuracy (Section 5.5). Furthermore, we constructed our dataset by adding data from different projects, repositories, and bug tracking applications; hence, our data represent different natures, kinds, and sizes of bugs, and the results so obtained on such unified and diverse data are accordingly satisfactory.

With regard to developer assignment, only the information of experts already assigned to a bug is used; attributes such as expertise, area, previously fixed bugs, etc. are not utilized. While this might affect the results, substantial performance improvement (superior to the state-of-the-art approaches) is this work in our case. Furthermore, we have emphasized throughout this work that involving too many unnecessary attributes can have an adverse impact on results. Therefore, if a certain level of performance bar is achieved with simple measures, adding unnecessary complexity may be inadvisable, as this may lead to conflicts.

Furthermore, conducting sample insertion in a self-paced manner requires human intervention to determine which samples are deemed "easy" and should be processed first; the model might behave differently for different sample selections. Nevertheless, we experimented with possible sample definitions (e.g. most and least frequent associations, highest and lowest weight tokens in associations) and retained the definition yielding the best results (most frequent associations, highest weight tokens).

7. Conclusions and future work

In this paper, a deep self-paced learning model is presented for bug classification and assigning of historical bug reports in open-source projects to the most appropriate developers. The staged model exploits LSTM for multi-class text classification, using a self-paced regularizer for data insertion, and finally attempts to assign the classified bug to the pertinent developer to be fixed. The model explores the practical implications of deep learning in software bug classification systems and bug triage, an area that had heretofore remained unexplored. More specifically, the impact of SPL combined with the deep learning methods is studied, and a deep self-paced integrated system is further proposed to improve the performance of existing systems.

This research work presents a novel bug classifier to solve the problem of categorizing bug reports pertaining to data of varying aspects and sizes. The framework presents an SPL model designed to identify bug types in historical bug reports related to open-source projects through classification of these bug reports. Such classification can further be used to speed up bug resolution and ensure correct assignment to pertinent developers with relevant expertise.

The SPAN exploits the easy and hard sample definitions for word associations that are fed into the model in a controlled manner, and accordingly exploits the natural phenomenon of learning from easy to hard concepts. Easy associations are fed first to the model, followed by the hard ones. The model improves the performance by up to 6% on various evaluation metrics when compared to our earlier work (SPBC [4]), and by a considerable amount relative to other state-of-the-art and baseline methods. Our model exhibits persistent and stable behavior (only a 1.2% drop in accuracy with 10-fold cross-validation) compared to the other baseline methods (4% to 30% drop in precision).

Data curation is a difficult task, and handling multiple features for a particular task is also cumbersome. Our research supports improving the basic task first or obtaining a superior result for a bigger task. Substantially better results can be obtained when data is introduced into a system in systematic fashion. Keeping this in mind, one potential avenue for future work would be to study the effect of SPL for runtime bug identification rather than historical bug reports. This could help in further improving the system by facilitating an understanding of the exact behavior involved in the real deployment of intelligent systems in the industry; hence, it could aid in securing the trust of practitioners and promoting their willingness to rely on automated systems for bug fixing and triage.

Moreover, this research will be extended to build a self-paced debug recommendation system that is capable of learning data from multiple domains and building its own easy and hard samples, allowing it to learn gradually from simple to complex bug fix solutions derived from heterogeneous sources. This will help in creating 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. **Shufeng Hao:** Methodology, Data curation, Writing – review & editing. **He Jiang:** Conceptualization, Methodology, Data curation, Writing – review & editing.

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.

Acknowledgments

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

[1] D. Matter, A. Kuhn, O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, in: Proc. 2009 6th IEEE Int. Work. Conf. Min. Softw. Repos. MSR 2009, 2009, http://dx.doi.org/10.1109/MSR.2009.5069491.

[2] W. Wu, W. Zhang, Y. Yang, Q. Wang, DREX: Developer recommendation with K-nearest-neighbor search and expertise ranking, in: Proc. - Asia-Pacific Softw. Eng. Conf. APSEC, 2011, http://dx.doi.org/10.1109/APSEC.2011.15.

[3] Y. Tian, D. Lo, C. Sun, Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, in: Proc. - Work. Conf. Reverse Eng. WCRE, 2012, http://dx.doi.org/10.1109/WCRE.2012.31.

[4] H. Mohsin, C. Shi, SPBC: A self-paced learning model for bug classification from historical repositories of open-source software, Expert Syst. Appl. 167 (2021) http://dx.doi.org/10.1016/j.eswa.2020.113808.

[5] Neelofar, M.Y. Javed, H. Mohsin, An automated approach for software bug classification, in: Proc. - 2012 6th Int. Conf. Complex, Intelligent, Softw. Intensive Syst, CISIS 2012, 2012, http://dx.doi.org/10.1109/CISIS.2012.132.

[6] H. Zhang, G. Zhong, Improving short text classification by learning vector representations of both words and hidden topics, Knowl.-Based Syst. 102 (2016) http://dx.doi.org/10.1016/j.knosys.2016.03.027.

[7] T. Zheng, L. Wang, Unlabeled text classification optimization algorithm based on active self-paced learning, in: Proceedings - 2018 IEEE International Conference on Big Data and Smart Computing, BigComp 2018, 2018, http://dx.doi.org/10.1109/BigComp.2018.00066.

[8] D. Cubranic, G.C. Murphy, Automatic bug triage using text categorization, in: 16th Int. Conference on Software Engineering and Knowledge Engineering, 2004.

[9] J. Xuan, H. Jiang, Z. Ren, J. Yan, Z. Luo, Automatic bug triage using semi-supervised text classification, in: SEKE 2010 - Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering, 2010.

[10] N. Limsettho, H. Hata, A. Monden, K. Matsumoto, Unsupervised bug report categorization using clustering and labeling algorithm, Int. J. Softw. Eng. Knowl. Eng. 26 (7) (2016) http://dx.doi.org/10.1142/S0218194016500352.

[11] Y. Tian, D. Lo, A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings, 2015, http://dx.doi.org/10.1109/SANER.2015.7081879.

[12] W. Zou, Y. Hu, J. Xuan, H. Jiang, Towards training set reduction for bug triage, in: Proceedings - International Computer Software and Applications Conference, 2011, http://dx.doi.org/10.1109/COMPSAC.2011.80.

[13] J. Liu, Y. Tian, X. Yu, Z. Yang, X. Jia, C. Ma, Z. Xu, A multi-source approach for bug triage, Int. J. Softw. Eng. Knowl. Eng. 26 (9–10) (2016) http://dx.doi.org/10.1142/S0218194016710030.

[14] R. Almhana, W. Mkaouer, M. Kessentini, A. Ouni, Recommending relevant classes for bug reports using multi-objective search, in: ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, http://dx.doi.org/10.1145/2970276.2970344.

[15] K. Somasundaram, G.C. Murphy, Automatic categorization of bug reports using latent Dirichlet allocation, in: Proceedings of the 5th India Software Engineering Conference, ISEC'12, 2012, http://dx.doi.org/10.1145/2134254.2134276.
[16] W. Zhang, S. Wang, Q. Wang, KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity, Inf. Softw. Technol. 70 (2016) http://dx.doi.org/10.1016/j.infsof.2015.10.004.
[17] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings - International Conference on Software Engineering, 14-22-MaAn Artificial Intelligence Paradigm for Troubleshooting Softwarey-2016, 2016, http://dx.doi.org/10.1145/2884781.2884804.
[18] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E.D. Trippe, J.B. Gutierrez, K. Kochut, A brief survey of text mining: Classification, clustering and extraction techniques, 2017, Retrieved from http://arxiv.org/abs/1707.02919.
[19] A. Elmishali, R. Stern, M. Kalech, An artificial intelligence paradigm for troubleshooting software bugs, Eng. Appl. Artif. Intell. 69 (2018) http://dx.doi.org/10.1016/j.engappai.2017.12.011.
[20] Y. Zhou, Y. Tong, R. Gu, H. Gall, Combining text mining and data mining for bug report classification, J. Softw.: Evol. Process 28 (3) (2016) http://dx.doi.org/10.1002/smr.1770.
[21] A. Sharma, S. Sharma, Bug report triaging using textual, Categorical Contextual Features using Latent DIRICHLET Allocation 1 (9) (2015) 85–96.
[22] I. Alazzam, A. Aleroud, Z. Al Latifah, G. Karabatis, Automatic bug triage in software systems using graph neighborhood relations for feature augmentation, IEEE Trans. Comput. Soc. Syst. 7 (5) (2020) http://dx.doi.org/10.1109/TCSS.2020.3017501.
[23] Q. Wang, Z. Mao, B. Wang, L. Guo, Knowledge graph embedding: A survey of approaches and applications, IEEE Trans. Knowl. Data Eng. 29 (12) (2017) 2724–2743, http://dx.doi.org/10.1109/TKDE.2017.2754499.
[24] C. Qin, H. Zhu, F. Zhuang, Q. Guo, Q. Zhang, L. Zhang, H. Xiong, A survey on knowledge graph-based recommender systems, Scientia Sinica Informationis 50 (7) (2020) 937–956, http://dx.doi.org/10.1360/SSI-2019-0274.
[25] X. Wang, X. He, Y. Cao, M. Liu, T.S. Chua, KGAT: Knowledge graph attention network for recommendation, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2019, pp. 950–958, http://dx.doi.org/10.1145/3292500.3330989.
[26] H.N. Tran, E. Cambria, A survey of graph processing on graphics processing units, J. Supercomput. 74 (5) (2018) 2086–2115, http://dx.doi.org/10.1007/S11227-017-2225-1.
[27] T. Dettmers, P. Minervini, P. Stenetorp, S. Riedel, Convolutional 2D knowledge graph embeddings, in: 32nd AAAI Conference on Artificial Intelligence, AAAI 2018, 2018.
[28] S. Ji, S. Pan, E. Cambria, P. Marttinen, P.S. Yu, A survey on knowledge graphs: Representation, acquisition, and applications, IEEE Trans. Neural Netw. Learn. Syst. (2021) http://dx.doi.org/10.1109/TNNLS.2021.3070843.
[29] D. Chen, B. Li, C. Zhou, X. Zhu, Automatically identifying bug entities and relations for bug analysis, in: IBF 2019-2019 IEEE 1st International Workshop on Intelligent Bug Fixing, 2019, http://dx.doi.org/10.1109/IBF.2019.8665494.
[30] C. Zhou, Intelligent bug fixing with software bug knowledge graph, in: ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, http://dx.doi.org/10.1145/3236024.3275428.
[31] S. Cavallari, E. Cambria, H. Cai, K.C.C. Chang, V.W. Zheng, Embedding both finite and infinite communities on graphs [application notes], IEEE Comput. Intell. Mag. 14 (2019) http://dx.doi.org/10.1109/MCI.2019.2919396.
[32] J. Anvik, G.C. Murphy, Reducing the effort of bug report triage: Recommenders for development-oriented decisions, ACM Trans. Softw. Eng. Methodol. 20 (3) (2011) http://dx.doi.org/10.1145/2000791.2000794.
[33] P. Bhattacharya, I. Neamtiu, C.R. Shelton, Automated, highly-accurate, bug assignment using machine learning and tossing graphs, J. Syst. Softw. 85 (10) (2012) http://dx.doi.org/10.1016/j.jss.2012.04.053.
[34] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: ESEC-FSE'09 - Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009, http://dx.doi.org/10.1145/1595696.1595715.
[35] R. Guzmán-Cabrera, M. Montes-Y-Gómez, P. Rosso, L. Villaseñor-Pineda, Using the web as corpus for self-training text categorization, Inf. Retr. 12 (3) (2009) http://dx.doi.org/10.1007/s10791-008-9083-7.
[36] S. Guo, R. Chen, M. Wei, H. Li, Y. Liu, Ensemble data reduction techniques and multi-RSMOTE via fuzzy integral for bug report classification, IEEE Access 6 (2018) 45934–45950, http://dx.doi.org/10.1109/ACCESS.2018.2865780.
[37] Y. Sui, X. Cheng, G. Zhang, H. Wang, Flow2Vec: Value-flow-based precise code embedding, Proc. ACM Program. Lang. 4 (2020) http://dx.doi.org/10.1145/3428301.
[38] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, W. Zhou, Familial clustering for weakly-labeled android malware using hybrid representation learning, IEEE Trans. Inf. Forensics Secur. 15 (2020) 3401–3414, http://dx.doi.org/10.1109/TIFS.2019.2947861.
[39] C. Li, F. Wei, J. Yan, X. Zhang, Q. Liu, H. Zha, A self-paced regularization framework for multilabel learning, IEEE Trans. Neural Netw. Learn. Syst. 29 (6) (2018) 2660–2666, http://dx.doi.org/10.1109/TNNLS.2017.2697767.
[40] X. Zhang, Y. Yao, Y. Wang, F. Xu, J. Lu, Exploring metadata in bug reports for bug localization, in: Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 2017-December, 2018, http://dx.doi.org/10.1109/APSEC.2017.39.