CNN-Based Automatic Prioritization of Bug Reports

Qasim Umer, Hui Liu, and Inam Illahi

***Abstract*—Software systems often receive a large number of bug reports. Triagers read through such reports and assign different priorities to different reports so that important and urgent bugs could be fixed on time. However, manual prioritization is tedious and time-consuming. To this end, in this article, we propose a convolutional neural network (CNN) based automatic approach to predict the multiclass priority for bug reports. First, we apply natural language processing (NLP) techniques to preprocess textual information of bug reports and covert the textual information into vectors based on the syntactic and semantic relationship of words within each bug report. Second, we perform the software engineering domain specific emotion analysis on bug reports and compute the emotion value for each of them using a software engineering domain repository. Finally, we train a CNN-based classifier that generates a suggested priority based on its input, i.e., vectored textual information and emotion values. To the best of our knowledge, it is the first CNN-based approach to bug report prioritization. We evaluate the proposed approach on open-source projects. Results of our cross-project evaluation suggest that the proposed approach significantly outperforms the state-of-the-art approaches and improves the average F1-score by more than 24%.**

***Index Terms*—Bug reports, deep learning, prioritization, reliability.**

I. Introduction

Software systems are often released with defects because of inadequate testing and system complexity [1]. Developers want feedback from users to resolve the defects that users experienced while using released systems. They employ issue reporting systems to collect feedback from users. Bugzilla [2], JIRA [3], and GitHub [4] are the most popular issue reporting systems. Users utilize such systems to report defects and track their progress. The utilization of issue tracking systems is standard practice in software development and maintenance [5] that helps developers to resolve reported defects. The resolution of reported defects has become an essential, expansive, and critical task in software maintenance due to the exponential growth of defects in complex software systems.

A bug report contains information that can be helpful in debugging and explains how exactly the product is crashed. A typical bug report includes predefined fields (e.g., product, priority, and severity), attachments (e.g., screenshots of the bug for developers), textual information (e.g., summary and description), and comments from the users and developers. Reporters provide all the required information while reporting bugs.

Software systems often receive a large number of bug reports [6]. Triagers read through such reports and assign different priorities to different reports so that important and urgent bugs could be fixed on time. Developers often do not fix bugs for years due to various constraints, e.g., time and availability of developers [7]. Therefore, triagers need to prioritize bug reports adequately so that developers can fix the ranked bugs in sequence. Different bug tracking systems have different priority levels for a reported bug. Therefore, the priority of bug reports is actually a multiobject classification. For example, in Bugzilla, the priority of a bug report can be defined from p_1 to p_5, where p_1 is the highest priority and p_5 is the lowest priority. Prioritizing bug reports is often a manual and time-consuming process. After a user reports a new bug through the bug tracking system, a triager is responsible first to examine the reported bug. Based on the examination, the triager decides its priority. The manual process of assigning priority increases the resolution time of the bug report [8]. To this end, some automated approaches have been proposed to suggest the priority of bug reports [5], [9]–[11]. However, the performance of such approaches deserves further significant improvement.

The machine learning and deep learning classifiers used for text classification have their own limitations for bug prioritization. Such as the best machine learning algorithm, support vector machine (SVM) reported by Umer *et al.* [9], requires feature modeling efforts. Long short-term memory (LSTM) does not extract the position-invariant features (similar features from the text that are similar in semantic but different in structure) as its output is position-variant dependent [12]. Consequently, LSTM does not identify the patterns (features) like *hate a lot* from the text [13]. In contrast to SVM and LSTM, convolutional neural network (CNN) not only extracts patterns and position-invariant features independently but also eliminates the feature modeling efforts for bug prioritization [14]. Notably, such pattern and position-invariant features are effective in *emotion analysis*. However, tuning hyperparameter settings to avoid overfitting problem is essential to CNN for optimal performance, which requires a deep understanding of both CNN and the problem to be resolved by CNN.

To this end, in this article, we propose a CNN-based automatic multiclass (*p1–p5*) prioritization for bug reports (*cPur*). Notably, we are the first to exploit CNN to bug report prioritization. We apply natural language processing (NLP) techniques to preprocess textual information of bug reports. From the

Manuscript received February 11, 2019; revised July 23, 2019 and October 9, 2019; accepted December 1, 2019. The work was supported by the National Natural Science Foundation of China under Grant 61690205 and Grant 61772071. Associate Editor: B. Xu. *(Corresponding author: Hui Liu.)*

The authors are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: qasimumer667@hotmail.com; liuhui08@bit.edu.cn; inamillahi@gmail.com).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2959624

preprocessed bug reports, we perform emotion analysis because hidden emotions may influence the priority of bug reports [9]. Users are rarely impassive when encountered with tiresome bugs. Consequently, the bug reports specified by such users may contain evident emotion that may reflect how urgent users want the bugs to be fixed. To classify the emotion of bug reports, we calculate the emotion of each bug report. Although emotion analysis has been leveraged for the prediction of bug priority [9], the proposed approach differs from existing work in that we compute emotions using a distributional semantic model [15] and train it on software engineering dataset. In contrast, Umer *et al.* [9] leverage a generic emotion repository *SentiWordNet*. A distributional semantic model [15] has been proved more effective than emotion repositories, and training it on software engineering specific dataset may significantly improve the accuracy in emotion computation for bug reports. Based on preprocessed textual information of bug reports, we construct a vector for each bug report with a *word2vector* model. We pass the constructed vector and the emotion of each bug report to a CNN-based classifier as input that predicts the priority. For the multiclass priority prediction, we train a CNN-based classifier. Finally, we evaluate the proposed approach on open-source projects. The results of the cross-project evaluation suggest that the proposed approach is accurate. On average, it improves the average F1-score upon state-of-the-art approaches by more than 24% (detailed results are provided in Section IV-E). Note that the reason to choose state-of-the-art approaches [1], [5], [9] as baselines is provided in Section IV-A.

This article makes the following contributions.

1) An automated CNN-based approach to suggest the priority of bug reports. To the best of our knowledge, it is the first CNN-based prioritization approach for bug reports.
2) Evaluation results of the proposed approach on the history data suggest that the proposed CNN-based approach is accurate in priority suggestion of bug reports and outperforms the state-of-the-art approaches.

The rest of this article is organized as follows. Section II discusses the related work. Section III defines the proposed approach details. Section IV describes the evaluation process of the proposed approach and its results. Section V explains the threats. Section VI concludes this article.

II. Related Work

A. Machine Learning Based Severity Identification/Prioritization of Bug Reports

Lamkanfi *et al.* [16] investigated whether the severity of a reported bug by analyzing its textual description using text mining algorithms can be accurately predicted. They applied a naive Bayes algorithm on the history data collected from the open-source community (Mozilla, Eclipse, and GNOME). They reported both precision and recall vary between 0.65–0.75 with Mozilla and Eclipse and 0.70–0.85 with GNOME.

Abdelmoez *et al.* [17] proposed an approach that uses naive Bayes classifier to predict the priority of bug reports. They used the data of four systems taken from three large open-source projects Mozilla, Eclipse, and GNOME. They prioritized the bug reports according to their mean time.

Tian *et al.* [18] proposed a novel approach leveraging information retrieval in a particular BM25-based document similarity function that automatically predicts the severity of bug reports. The proposed approach automatically analyzes bug reports and focuses on predicting fine-grained severity labels, namely the different severity labels of Bugzilla including blocker, critical, major, minor, and trivial. Results suggest that fine-grained severity prediction outperforms the state-of-the-art study and brings significant improvement.

Tian *et al.* [5] proposed an automated classification approach (*DRONE*) for priority prediction of bug reports. They employed linear regression (LR) for priority classification and achieved the average F1-score up to 29%.

Alenezi and Banitaan [8] adopted naive Bayes, decision tree, and random forest [8] to execute the priority prediction. They used two feature sets, i.e., 1) based on *TF* weighted words of bug reports, and 2) based on the classification of bug reports attributes. Evaluation results suggest that the usage of the second feature set performed better than the first feature set, where random forests and decision trees outperform naive Bayes.

Tian *et al.* [1] predicted the priority of bug reports using the nearest neighbor approach to identify fine-grained bug report labels. They applied the proposed approach to a larger collection of bug reports consisting of more than 65 000 Bugzilla reports.

Tian *et al.* [19] used three open-source software systems (OpenOffice, Mozilla, and Eclipse) and found that around 51% of the duplicate bug reports have inconsistent human-assigned severity labels even though they refer to the same software problem. Results suggest that current automated approaches perform well and their agreement varies from 77% to 86% with human-assigned severity labels.

Choudhary [20] recently developed a model for priority prediction using a SVM that assigns priorities to Firefox crash reports in the Mozilla Socorro server based on the frequency and entropy of the crashes.

As a conclusion, researchers have proposed a number of machine learning approaches to predict the priority of bug reports. Our proposed approach differs from the existing approaches in that we are first to apply CNN-based prioritization of bug reports.

B. Deep Learning

Deep learning is an emerging machine learning technique that enables a machine to analyze complex and abstract data features through hierarchical neural networks. Lately, deep learning is getting a lot of attention and attaining results sometimes better than humans. It has achieved significant results in the field of computer vision [21], speech recognition [22], and sentiment analysis [23].

Deep learning is also increasingly prevalent in the field of software engineering and playing an important role in software engineering tasks, e.g., development, testing, and maintenance [24]. According to Li *et al.*, different studies employed deep learning techniques for the maintenance of softwares in

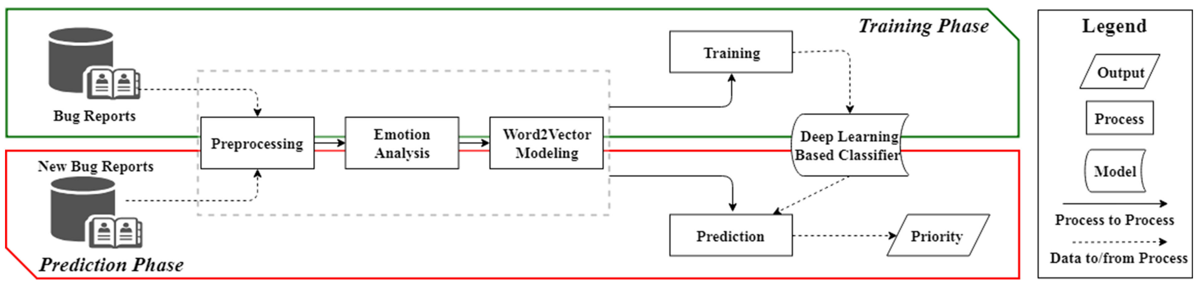


Fig. 1. Overview of the proposed approach.

which some of them are related to bug handling, e.g., bug localization, bug report summarization, bug triager, and duplicate bug detection [24]. The successful applications of deep learning techniques to such software engineering tasks inspire us to apply deep learning techniques, e.g., CNN, to prioritization of bug reports.

C. Emotion Analysis

Emotion analysis have achieved excellent results in semantic parsing, search query retrieval [25], sentence modeling [26], traditional NLP tasks [27], and sentiment analysis [23]. Some state-of-the-art classification approaches based on emotion analysis are discussed in the following.

Ouyang *et al.* [23] used a recurrent neural network to classify Italian Twitter messages for predicting emotions. The work is built upon a deep learning approach. They leveraged large amounts of weakly labeled data to train a two-layer CNN. To train their network, they applied a form of multitask training. Their work participated in the EvalItalia-2016 competition and outperformed all other approaches to the sentiment analysis task.

Umer *et al.* [9] recently proposed an emotion-based automatic approach (eApp) for the priority classification. They employed emotion analysis for priority classification and used the SVM (a machine learning algorithm) for the prioritization of bug reports. Results suggest that the proposed approach outperforms the state-of-the-art approach and improves F1-score by more than 24%. Our proposed approach also employs the emotion analysis for priority classification but differs from their approach in that we exploit a distributional semantic model to compute emotions of bug reports as an essential step for priority classification.

Other different studies related to bug reports include detection of bug report duplication [28]–[30], a recommendation of an appropriate developer to a new bug report [31]–[33], and the prediction of bug fixing time [34]–[36].

III. Approach

A. Overview

An overview of the CNN-based prioritization for bug reports (cPur) is presented in Fig. 1. The proposed approach recommends a priority level for each bug report as follows.

1) First, we collect the history data of bug reports of open-source projects as training data.
2) Second, we apply NLP techniques to the bug reports for preprocessing.
3) Third, we perform emotion analysis on bug reports and compute the emotion of each bug report.
4) Fourth, we create a vector for each bug report by using its preprocessed words.
5) Finally, we train a CNN-based classifier for the priority prediction. We pass the generated vector and the emotion of each bug report to the classifier as input that predicts the priority of bug reports.

We introduce each of the key steps of the proposed approach in the following sections.

B. Illustrating Example

We use the following example to illustrate how the proposed approach prioritizes bug reports. It is an Android bug report (81613) collected from Google Issue Tracker [37]. It was created on December 3, 2014, and closed on December 3, 2014.

1) Product = "*Android Studio*" is the name of the product that is affected by the bug.
2) Textual Information = "*First run wizard delete SDK if androidsdk.repo and androidsdk.dir point to the same dir*" explains the bug. It may contain information on bug regeneration.
3) Priority = "p_1" is a priority of the example bug report that could be left blank during reporting bugs. A triager may then assign the priority to the bug report.

We present the details on how the proposed approach works for the illustrating example in the following section.

C. Problem Definition

A bug report r from a set of bug reports $\mathbb{R}$ can be formalized as

$$r = < t, p > \tag{1}$$

where, t is the textual information of r and p is an assigned priority to r. For the illustrating example presented in Section III-B, we have

$$r_e = < t_e, p_e > \tag{2}$$

where $t_e =$ *"First run wizard delete SDK if androidsdk.repo and androidsdk.dir point to the same dir,"* and $p_e = p_1$

The proposed approach suggests the priority of the new bug report as either p_1, p_2, p_3, p_4, or p_5, where p_1 is the highest priority and p_5 is the lowest priority. Consequently, the automatic prioritization of a new bug report r could be defined a mapping f

$$f : r \rightarrow c$$
$$c \in \{p_1, p_2, p_3, p_4, p_5\}, \quad r \in \mathbb{R} \tag{3}$$

where c is a suggested priority from a priority set (p_1, p_2, p_3, p_4, p_5).

D. Preprocessing

Bug reports contain irrelevant and unwanted text, e.g., punctuation. The input of irrelevant text to the classification algorithms in an overhead as it increases the processing time and utilizes more memory for processing. Therefore, we perform preprocessing to increase the performance of the proposed approach and to make it cost effective. NLP techniques are often used for the preprocessing of bug reports that include tokenization, stop-word removal, negation handling, spell correction, modifier word recognition, word inflection, and lemmatization. We employ the following preprocessing steps to clean the textual information of bug reports.

1) *Tokenization:* The text of bug reports often contains words and special characters, e.g., spaces and punctuation marks. Tokenization removes the special characters and decomposes the text into words (tokens).
2) *Stop-word removal:* Textual documents often contain words that are used to make sentences meaningful but do not have meaning individually. Such words are known as stop-words. We remove such words from the extracted words in tokenization. Note that bug reports contain some programming-related words; however, we only remove the English language stop-words from the text.
3) *Spell correction:* Users type the unstructured fields while reporting bugs, e.g., textual information: summary and description that may have spelling mistakes. Therefore, we apply an automated way to correct spelling mistakes.
4) *Negation and modifiers:* The usage of negation or modifier with an English word changes its impact in the sentence. For example; *good* and *not good* are reciprocal to each other. Similarly, *good* and *very good* have different intensities. We apply negation and word modifier recognition during the calculation of emotion of each bug report, as mentioned in Section III-E.
5) *Word inflection and lemmatization:* Word inflection converts the words into their singular form. For example; inflection converts the word *errors* into *error*. Whereas, lemmatization converts comparative and superlative words into their base words. For example, lemmatization converts the word *crashed* into *crash*. We apply both word inflection and lemmatization on the extracted words and finally fold them into lowercase.

To perform preprocessing, we utilize *Python Natural Language Toolkit (NLTK)* [38] and *Python TextBlob Library* [39]. After preprocessing, a bug report r can be represented as

$$r' = \langle ws, p \rangle \tag{4}$$

TABLE I
EXAMPLE OF PREPROCESSING AND EMOTION ANALYSIS

Before Preprocessing	**After Preprocessing**
First run wizard deletes SDK if androidsdk.repo and androidsdk.dir point to the same dir	wizard delete sdk androidsdkrepo androidsdkdir point dir

$$ws = \langle w_1, w_2, \ldots, w_n \rangle \tag{5}$$

where $w_1, w_2, \ldots, w_n$ are the words (tokens) from the textual description of r after preprocessing.

For the illustrating example presented in Section III-B, the second column of Table I presents the preprocessing results of an example bug report r_e. After preprocessing, we have

$$r'_e = \langle \text{wizard}, \text{delete}, \text{sdk}, \ldots, \text{dir}, p_1 \rangle \tag{6}$$

where wizard, delete, sdk, ..., dir are the preprocessed words from r_e.

E. Emotion Analysis

Users are rarely impassive when encountered with tiresome bugs. As a result, the bug reports specified by such users may contain evident emotion. For example, the bug report *5083: Breakpoint not hit* has negative emotion due to a word *hit*. Whereas the bug report *8423: Thank you, that was really helpful. "I want them to resize based on the length of the data they're showing."* has positive emotion due to words *Thank and helpful*.

To classify the emotion of bug reports whether the emotion of the reporter in the bug reports is positive or negative, we calculate the emotion of each bug report. There are many repositories for the emotion analysis of text documents, e.g., SentiWordNet [40]. However, to the best of our knowledge, *SentiStrengthSE* [41], *SentiCR* [42], *Senti4SD* [15], and *EmoTxt* [43] are the repositories used for emotion analysis of software engineering text. We choose *Senti4SD* for emotion analysis because it is commonly used repository and outperforms the *SentiStrength*, *SentiStrengthSE*, and *SentiCR* for the text classification in the software engineering domain [15]. We input each bug report to the distributional semantic model [15] to compute its emotion. The distributional semantic model creates a mathematical point in high-dimensional vector space to represent words. It depends on the distributional hypothesis believing that semantically similar words belong to the same context [15]. It returns the emotion of a given bug report based on its emotion words, negation, and modifiers. We store the computed emotions with the corresponding bug reports. After emotion analysis, a bug report can be represented as

$$r'' = \langle e, w_1, w_2, \ldots .w_n, p \rangle \tag{7}$$

where e is the emotion of r'.

For the motivating example presented in Section III-B, we input the preprocessed text of the example bug report to the

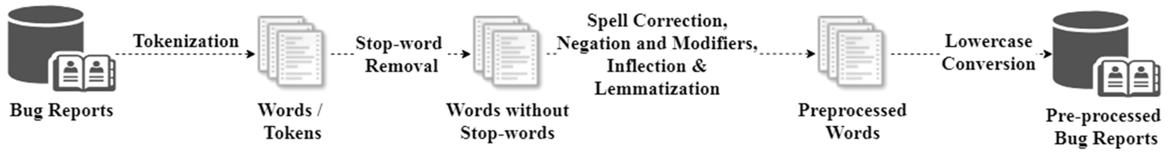


Fig. 2. Overview of preprocessing.

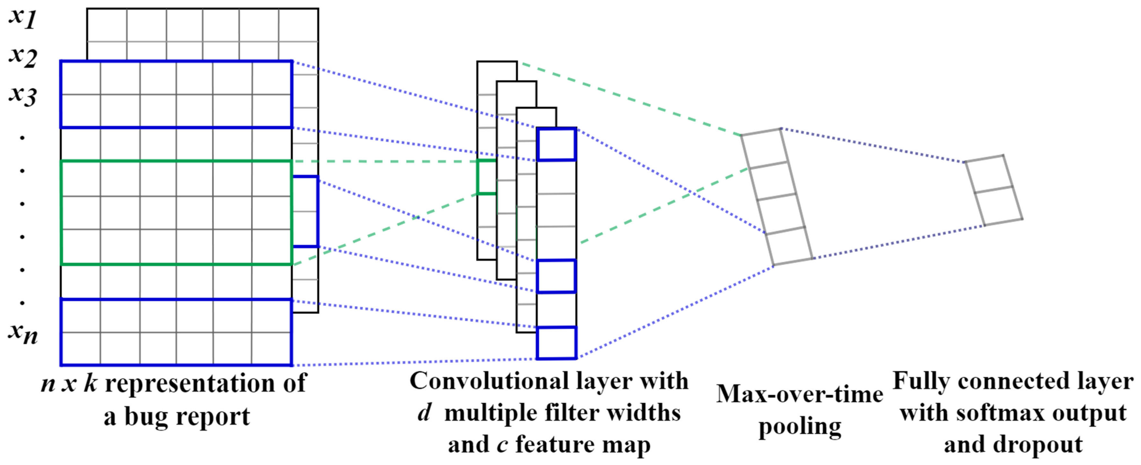

x1
x2
x3
xn
n x k representation of a bug report
Convolutional layer with d multiple filter widths and c feature map
Max-over-time pooling
Fully connected layer with softmax output and dropout

Fig. 3. Overview of the priority prediction model.

Senti4SD and we have

$$r''_e = \langle \text{positive}, \text{wizard}, \text{delete}, \text{sdk}, \ldots, \text{dir}, p_1 \rangle \quad (8)$$

where positive is the calculated emotion of the example bug report.

F. Word2Vector Modeling

In this step of the automatic prioritization approach of bug reports, we construct a vector for each bug report. We pass the preprocessed words $w_1, w_2, \ldots, w_n$ from (7) to a *skip-gram-based word2vector model* [44]. It is an efficient method for learning continuous word representation (a high-quality distributed vector) based on a single hidden layer neural network. The model captures a large number of precise syntactic and semantic word relationships and returns a k-dimensional vector.

For the motivating example presented in Section III-B, the preprocessed words wizard, delete, sdk, ..., dir are passed to the skip-gram model to convert them into a vector. For example, wizard is presented as [0.102, −4.31, −0.003, ...].

G. Priority Prediction Model

The proposed approach exploits the CNN to relate a vector and an emotion with the priority of a bug report. We use a CNN to predict the priority of bug reports for the following reasons. First, the CNN uses the vector concatenation method to concatenate incoming inputs into one long input vector. Consequently, the CNN can handle the long-term dependencies better than the recurrent neural network. Second, the usage of different sized filters convolution with the long input vector enables the CNN to encode short-term and long-term dependencies by small and large filter sizes, respectively. Third, the CNN does not suffer from the exploding gradient problem of a recurrent neural network [45] by using different filter sizes.

1) Overview: The overview of the proposed model is shown in Fig. 3. Given a bug report r with its emotion e (computed in Section III-E) and k-dimensional vector $\mathbf{x}$ (constructed in Section III-F), a bug report of maximal length of n (to compute n, we first find the preprocessed bug reports with maximum length and apply padding on the remaining bug reports) can be represented as $\mathbf{x}$ (i.e., the input of the CNN in Fig. 3), which is calculated as

$$\mathbf{x} = \langle e, u_1, u_2, \ldots., u_n \rangle \quad (9)$$

$$\mathbf{x}' = \langle v_1, v_2, v_3, \ldots., v_n \rangle \quad (10)$$

where v_i is the vector representation of e and w_i.

2) Filter Operation: The CNN applies a filter $\mathbf{w} \epsilon \, \mathbb{R}^{dk}$ to a window of d words to generate a new feature. For instance, a new feature c_i is generated from a window of words $v_{i,i+d-1}$ that can be formalized as

$$c_i = f(\mathbf{w}.v_{i,i+d-1} + b) \quad (11)$$

where b represents a bias term that belongs to $\mathbb{R}$, and f is a hyperbolic tangent nonlinear function. This filter generates a feature map using each window of the features $< v_{1:d}, v_{2:d+1}, \ldots, v_{n-d+1:n} >$. A generated feature map $\mathbf{c}$ that belongs to $\mathbb{R}^{n-d+1}$ can be formalized as

$$\mathbf{c} = \langle c_1, c_2, \ldots, c_{n-d+1} \rangle. \tag{12}$$

For the motivating example presented in Section III-B, we first feed inputs (emotion and preprocessed text) into an embedding layer that converts them into numerical vectors. Second, we pass the numerical vectors into a CNN with *dropout* = 0.2. Because we observe that this setting results in the minimum loss in the training phase when the dropout varies from 0.0 to 0.5 (where the step size is 0.1). Notably, we include dropout to prevent overfitting. A fully connected softmax layer involves most of the parameters. Thus, neurons create dependencies between each other that restrict the individual power of each neuron leading to overfitting of training data. Consequently, overfitting decreases the performance of the model. Finally, we use three layers of the CNN. Notably, we use three convolutional layers because each convolution generates tensors of different shapes due to multiple filters. We create a layer for each of tensors to iterate through them for merging the results into one big feature vector. We forward the output of the CNN to a flatten layer that converts the numerical vectors into a one-dimensional vector. We apply the different length of filters (3, 4, and 5) on each n x k vector and generate new features.

3) Pooling Operation: A max-over-time pooling operation is applied on the feature map to get the maximum value $\hat{c}$ from (12). The pooling operation helps to find the most important features from its feature map, i.e., the features having the highest value.

The proposed model applies multiple different size filters and extracts one feature from one filter. Such features construct the penultimate layer and are forwarded to a fully connected softmax layer. The output of the softmax layer is the probability distribution of the priority levels.

IV. Evaluation

In this section, the performance of the proposed approach is evaluated on the bug reports of four open-source Eclipse projects.

A. Research Questions

The evaluation investigates the following research questions.

1) RQ1: Does *cPur* outperform the state-of-the-art approaches in prediction of bug reports?
2) RQ2: How does different input (text and emotion) influence the performance of *cPur*?
3) RQ3: How does preprocessing influence the performance of *cPur*?
4) RQ4: How does the length of filters influence the performance of *cPur*?
5) RQ5: How does the training size influence the performance of *cPur*?
6) RQ6: Does convolution neural network outperform other classification algorithms (traditional machine learning algorithm and LSTM) in predicting priority of bug reports?

The first research question (RQ1) examines the performance improvement of *cPur* against the state-of-the-art approaches. To this end, we select three approaches *eApp* (proposed by Umer *et al.* [9]), *DRONE* (proposed by Tian *et al.* [5]), and *DRONE** (proposed by Tian *et al.* [1]) for the comparison because of the following reasons. First, *eApp*, *DRONE*, and *DRONE** were designed for automatic prioritization of bug reports as our approach is. Second, they are recently proposed and represent the state of the art.

The second research question (RQ2) investigates the influence of the given inputs. We provide two inputs (preprocessed text of bug reports and their emotion analysis results) to the CNN-based approach for the priority prediction of bug reports. We want to know to what extent does this affect the performance, respectively.

The third research question (RQ3) investigates the impact of the preprocessing on the performance of *cPur*. Most of the textual datasets are not clean, i.e., they may contain punctuation. Therefore, we perform preprocessing (mentioned in Section III-D) to clean the given dataset.

The fourth research question (RQ4) investigates the impact of the length of filters on the effectiveness of the proposed approach.

The fifth research question (RQ5) investigates the relationship between the training size and the effect of the proposed approach.

The sixth research question (RQ6) compares the selected classification algorithm (*CNN*) with alternatives. We choose *SVM* because Umer *el al.* [9] recently declared it as a best machine learning algorithm for priority prediction. Whereas, we select *LSTM* because it is approved effective in NLP [46].

B. Dataset

We exploit the dataset created by Tian *et al.* [5] and reused by Tian *et al.* [1] and Umer *et al.* [9]. They investigated the bug repository of Eclipse, which is an open-source integrated development platform. They collected the bug reports submitted from October 2001 to December 2007 from Bugzilla [47]. Notably, they only collected the defect reports and ignored the enhancement reports. The resulting dataset includes the bug reports of four open-source projects: Java development tools (JDT), Eclipse's C/C++ Development Tooling (CDT), Plug-in Development Environment (PDE), and Platform. Their summary attribute defines the reported bugs, whereas their priority attribute indicates their importance and urgency. The total number of bug reports in the dataset are 80 000 in which 25%, 28%, 16%, and 31% of bug reports belong to CDT, JDT, PDE, and Platform, respectively. This dataset is also used by Umer *et al.* and Tian *et al.* to evaluate their approaches that are selected in this article as a baseline approaches.

C. Process

We evaluate the proposed approach as follows. First, we reuse the bug reports $\mathbb{R}$ of four open-source projects from Bugzilla and apply NLP techniques to preprocess them, as mentioned in Section III-D. Second, we carry out a cross-project validation on $\mathbb{R}$. We partition $\mathbb{R}$ dataset into four sets based on the project notated as $S_i (i = 1 \ldots 4)$. For the ith cross validation, we consider all bug reports except for those in S_i as a training dataset and treat the bug reports in S_i as a testing dataset. For the ith cross validation, the evaluation process as follows.

1) First, we select all *training reports* (TR) from training dataset that is a union of all sets but S_i, calculated as

$$\text{TR}_i = \bigcup_{i\in[1,4]\ \wedge\ j\neq i} S_j. \tag{13}$$

2) Second, we train a LSTM with data from TR.
3) Third, we train a CNN with data from TR.
4) Fourth, we train the machine learning algorithms (SVM [9] and LR [1], [5]) with data from TR.
5) Fifth, for each report in S_i, we predict the priority of each bug report using the trained LSTM, CNN, LR, and SVM to compare their status with its real priority.
6) Finally, we compute the evaluation metrics for each algorithm to compare their performances.

D. Metrics

Given the bug reports $\mathbb{R}$, the performance of the proposed approach is evaluated by calculating the priority specific precision P, recall R, and F1-score $F1$ as

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{14}$$

$$R = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{15}$$

$$F1 = \frac{2 * P * R}{P + R} \tag{16}$$

where P, R, and $F1$ are, respectively, precision, recall, and F1-score of the approaches for priority prediction of $\mathbb{R}$ whose actual priority is P_i. TP is the number of $\mathbb{R}$ that are truly predicted as P_i, FP is the number of $\mathbb{R}$ that are falsely predicted as P_i, and FN is the number of $\mathbb{R}$ that are not predicted as P_i but they are actually P_i.

Our multiclass classification problem has five priority classes (levels) as labels. Therefore, we also perform macroanalysis and microanalysis for all priority levels *C*, which are commonly used to evaluate the performance of multiclass classification [9], [48]. Where macroanalysis combines precision and recall of multiclass priority levels by averaging their values. It simply normalizes the sum of precision of each of the priority levels using the number of different values. Whereas, microanalysis has a similar idea to macroanalysis but computes precision and recall from the sum of true positive, true negative, false positive, and false negative values of all priority levels. In contrast to macroanalysis, microanalysis takes the frequency of each priority level into consideration. We calculate macroprecision P_{mac}, macrorecall R_{mac}, macro F1-score $F1_{\text{mac}}$, microprecision P_{mic}, microrecall R_{mic}, and micro F1-score $F1_{\text{mic}}$ as follows:

$$P_{\text{mac}} = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i} \tag{17}$$

$$R_{\text{mac}} = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i} \tag{18}$$

$$F1_{\text{mac}} = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{2 * P_{\text{mac}} * R_{\text{mac}}}{P_{\text{mac}} + R_{\text{mac}}} \tag{19}$$

$$P_{\text{mic}} = \sum_{i=1}^{|C|}\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i} \tag{20}$$

$$R_{\text{mic}} = \sum_{i=1}^{|C|}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i} \tag{21}$$

$$F1_{\text{mic}} = \sum_{i=1}^{|C|}\frac{2 * P_{\text{mic}} * R_{\text{mic}}}{P_{\text{mic}} + R_{\text{mic}}}. \tag{22}$$

We also compute the Hamming-loss error. It calculates the average number of the relevance of a bug report to a priority level, which is falsely predicted [49]. It normalizes the loss over a total number of priority levels and the total number of bug reports using priority prediction error (an incorrect priority level is predicted) and missing error (a relevant priority level is not predicted). The Hamming-loss error HE can be formalized as

$$\text{HE} = \frac{1}{|N|\,.\,|P|}\sum_{i=1}^{|N|}\sum_{j=1}^{|P|}(y_{i,j}, z_{i,j}) \tag{23}$$

where *N* is a number of bug reports, *P* is a number of priority levels, $y_{i,j}$ is the true priority levels, and $z_{i,j}$ is the predicted priority levels.

E. RQ1: Comparison Against the State-of-the-Art Approaches

To answer the research question **RQ1**, we compare the *cPur* against the state-of-the-art approaches (*eApp*, *DRONE**, and *DRONE*) in priority prediction of bug reports. To this end, we perform microanalysis and macroanalysis to find out the performance improvement of *cPur* against each class. We also conduct the priority-level and project-level comparison to evaluate the performance improvement of *cPur* for each priority and each project, respectively.

1) Comparison on Microanalysis and Macroanalysis: Evaluation results of microanalysis and macroanalysis are presented in Table II. The first column presents the approaches. Columns 2–4 and 5–7 present the performance in microanalysis and macroanalysis, respectively. The last column presents the error. The rows of the table present the performance results of *cPur*, *eApp*, *DRONE**, and *DRONE*, respectively.

From Table II, we make the following observations.

1) *cPur* outperforms *DRONE** and *DRONE* in both macroanalysis and microanalysis. It indicates that *cPur* improves F1-score not only for all priority levels (as whole system) but also for each priority level (individual). However, we also notice that *cPur* outperforms *eApp* in macroanalysis only, and its precision in microanalysis is slightly lower than that of *eApp*.
2) The performance improvement of *cPur* upon *eApp* in F1-score for macroanalysis and microanalysis are 22.94%

TABLE II
PERFORMANCE ON MICRO AND MACRO LEVELS

	Macro			Micro			Error
	Precision	Recall	F1-score	Precision	Recall	F1-score	
cPur	**57.80%**	**56.21%**	**56.99%**	**58.79%**	**41.98%**	**48.98%**	**0.4291**
eApp	57.62%	38.78%	46.36%	58.94%	38.14%	46.31%	0.4397
DRONE*	55.27%	38.57%	45.43%	56.11%	35.49%	43.48%	0.4626
DRONE	53.63%	32.23%	40.26%	54.45%	31.77%	40.13%	0.4793

TABLE III
PERFORMANCE ON PRIORITY LEVEL

	P1			P2			P3			P4			P5		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
cPur	**71.51%**	**32.39%**	**44.58%**	**75.28%**	**23.61%**	**35.94%**	**60.17%**	**95.46%**	**73.81%**	**79.26%**	**22.86%**	**35.49%**	**67.18%**	**35.18%**	**46.18%**
eApp	59.42%	26.49%	36.64%	58.19%	22.75%	32.71%	57.75%	91.38%	70.77%	60.89%	22.86%	33.24%	58.46%	27.21%	37.14%
DRONE*	57.76%	21.51%	31.35%	54.32%	17.95%	26.98%	55.79%	93.87%	69.99%	56.05%	17.41%	26.57%	53.82%	22.59%	31.82%
DRONE	56.35%	18.39%	27.73%	53.83%	14.32%	22.54%	54.54%	93.23%	68.82%	54.94%	13.67%	21.89%	52.59%	19.22%	28.15%

TABLE IV
PERFORMANCE ON PROJECT LEVEL

	CDT			JDT			PDE			Platform		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
cPur	**67.20%**	**60.44%**	**63.64%**	**63.70%**	**60.52%**	**62.07%**	**74.63%**	**72.34%**	**73.47%**	**59.92%**	**55.58%**	**57.67%**
eApp	60.99%	40.69%	48.81%	59.86%	41.24%	48.84%	64.22%	34.79%	45.13%	50.70%	35.83%	41.99%
DRONE*	57.02%	35.38%	43.67%	55.93%	38.09%	45.32%	59.47%	31.39%	41.09%	48.14%	31.72%	38.24%
DRONE	56.66%	32.74%	41.50%	54.79%	35.21%	42.87%	58.26%	29.22%	38.92%	48.09%	29.89%	38.86%

= (56.99%–46.36%)/46.36% and 5.77% = (48.98%–46.31%)/46.31%, respectively. Moreover, the performance improvement of *cPur* upon *DRONE** in F1-score for macroanalysis and microanalysis are 25.45% = (56.99%–45.43%)/45.43% and 12.65% = (48.98%–43.48%)/43.48%, respectively. Similarly, the performance improvement of *cPur* upon *DRONE* in F1-score for macroanalysis and microanalysis are 41.55% = (56.99%–40.26%)/40.26% and 22.07% = (48.98%–40.13%)/40.13%, respectively.

3) *cPur* reduces the error upon *eApp*, *DRONE**, and *DRONE* by 2.5% = (0.4397–0.4291)/0.4291, 7.81% = (0.4626–0.4291)/0.4291, and 11.6% = (0.4790–0.4291)/0.4291, respectively.

2) Comparison on Priority Level and Project Level: Evaluation results of each priority and each project for the involved approaches are presented in Tables III and IV, respectively. Notably, we apply tenfold cross validation and cross-project evaluation techniques to produce the priority level and project level results, respectively. Therefore, the reproduced results of the state-of-the-art approaches may differ from the originally reported results (using n-cross validation).

In Table III, the first column presents the approaches. Columns 2–4, 5–7, 8–10, 11–13, and 14–16 present the performance of priority p_1, p_2, p_3, p_4, and p_5, respectively. The rows of the table present the performance results of *cPur*, *eApp*, *DRONE**, and *DRONE*, respectively.

In Table IV, the first column presents the approaches. Columns 2–4, 5–7, 8–10, and 11–13 present the performance of projects CDT, JDT, PDE, and Platform, respectively. The rows of the table present the performance results of *cPur*, *eApp*, *DRONE**, and *DRONE*, respectively.

From Tables III and IV, we make the following observations.

1) On each priority level, *cPur* outperforms *eApp*, *DRONE**, and *DRONE*. The improvement of *cPur* upon *eApp* in F1-score varies from 4.30% = (73.81%–70.77%)/70.77% to 24.34% = (46.18%–37.14%)/37.14%. Moreover, the improvement of *cPur* upon *DRONE** in F1-score varies from 5.46% = (73.81%–69.99%)/69.99% to 45.13% = (46.18%–31.82%)/31.82%. Similarly, the improvement of *cPur* upon *DRONE* in F1-score varies from 7.25% = (73.81%–68.82%)/68.82% to 64.04% = (46.18%–28.15%)/28.15%.
2) On each project level, *cPur* outperforms *eApp*, *DRONE**, and *DRONE*. The improvement of *cPur* upon *eApp* in F1-score varies from 27.10% = (62.07%–48.84%)/48.84% to 62.79% = (73.47%–45.13%)/45.13%. Moreover, the improvement of *cPur* upon *DRONE** in F1-score varies from 31.24% = (62.07%–45.32%)/45.32% to 78.80% = (73.47%–41.09%)/41.09%. Similarly, the improvement of *cPur* upon *DRONE* in F1-score varies from 44.79% = (62.07%–42.87%)/42.87% to 88.75% = (73.47%–38.92%)/38.92%.

To validate the significant difference among *cPur*, *eApp*, *DRONE**, and *DRONE*, we employ one-way analysis of variance (ANOVA). ANOVA determines whether there are any statistically significant difference between the means of independent (unrelated) groups [50], where the unit of analysis in ANOVA is a project. ANOVA is employed because all approaches are applied to the same projects. It may validate whether the only difference (single factor, i.e., different approaches) leads to the difference in performance. We compute ANOVA on Excel with its default settings and do not involve any adjustment. Notably, ANOVA on F1-score is conducted independently, where the unit

TABLE V
ANOVA Analysis on F1-Score

Source of Variation	SS	df	MS	F	P_{value}	F_{crit}
F1-score						
Between Groups	0.057071	2	0.028535	10.820254	0.004034	4.256495
Within Groups	0.023735	9	0.002637			
Total	0.080805	11				

of analysis is a project. Table V describes the results of ANOVA analysis, which presents $F > F_{\text{cric}}$ and p-value < (alpha = 0.05) are true for F1-score, where $F = 10.82$, $F_{\text{cric}} = 4.26$, and p-value $= 0.004$. It suggests that the factor (using different approaches) has a significant difference in F1-score.

Moreover, we perform *Wilcoxon test* (using *Stata* software built-in settings) to calculate the difference between approaches and analyze these differences. The results present p-value < (alpha = 0.05) is true for F1-score, where P-value $= 0.02$.

Furthermore, we quantify the *effect size* to check the difference between approaches by employing Cohen's delta d, where $d >= 0.2$, $d >= 0.5$, and $d >= 0.8$ represents the difference as *small*, *medium*, and *large*, respectively. Result ($d = 0.75$) suggests that the difference between approaches is *medium*.

Finally, we compute the time cost of preprocessing, emotion analysis, Word2Vector modeling, training, and testing processes to investigate the efficiency of *cPur*. The results suggest that *cPur* is efficient. The average time cost of the preprocessing, emotion analysis, and Word2Vector modeling is 2.01 min, 5.67 min, and 2.98 min, respectively. Notably, parts-of-speech tagging significantly increases the time cost of emotion analysis. Moreover, the training time cost of *cPur* (4.23 min) is higher than *DRONE** (3.52 min), *DRONE* (3.52 min), and *eApp* (3.98 min), respectively. However, using the trained models, *cPur* takes 1.06 min in priority prediction, which is equal to *DRONE** and *DRONE*, and faster than *eApp* by 0.03 min. On average, the training process for CDT, JDT, PDE, and Platform requires 0.8 min, 1.2 min, 1.33 min, and 0.9 min, respectively. The testing times of CDT, JDT, PDE, and platform are 0.22 min, 0.26 min, 0.38 min, and 0.21 min, respectively.

Based on the preceding analysis, we conclude that *cPur* achieves a significant improvement upon the state-of-the-art approaches.

F. RQ2: Influence of Different Inputs

To answer the research question RQ2, we perform a project-level comparison with and without different inputs (textual features of bug reports and their emotions).

Evaluation results of the proposed approach by enabling and disabling different inputs are presented in Table VI. The first column presents the input settings. Columns 2–4, 5–7, 8–10, and 11–13 present the performance of project CDT, JDT, PDE, and Platform, respectively. The performance of *cPur* upon different settings is presented in the rows of the table. Fig. 4 also visualizes the performance difference of the proposed approach upon different settings.

From Table VI and Fig. 4, we make the following observations.

1) Disabling emotion value from the input significantly reduces the recall and F1-score of the proposed approach. The decrease in recall varies from 3.45% = (60.52%–58.43%)/60.52% to 7.66% = (72.34%–66.80%)/72.34%. The decrease in F1-score varies from 1.52% = (57.67%–56.80%)/56.80% to 6.42% = (73.47%–69.03%)/69.03%.
2) Disabling textual features from the input also significantly reduces recall and F1-score of the proposed approach. The decrease in recall varies from 7.63% = (60.52%–55.90%)/60.52% to 13.66% = (72.34%–62.46%)/72.34%. The decrease in F1-score varies from 6.32% = (62.07%–58.38%)/58.38% to 10.22% = (73.47%–66.66%)/66.66%.
3) Disabling either emotion value or textual features from the input would decrease the precision on most cases (on three out of four subject applications). However, it results in slight increase in precision on Platform. The increase is 0.53% = 60.45%–59.92% (disabling emotion) and 1.16% = 61.08%–59.92% (disabling textual features), respectively. However, due to the poor interpretability of deep neural networks, we have not yet fully understood the rationale for the increase of precision on Platform caused by disabling emotion value or textual features. Overall, disabling either the emotion value or textual features from the input has little and inconsistent influence on the precision of the proposed approach.

To investigate the existence of emotion and why emotion features work in prioritization of bug reports, we randomly select 200 sample reports (50% positive and 50% negative) and manually check the effect of emotion features to the proposed approach. The manual checking is accomplished by five software engineering professionals. Two of them are Ph.D. scholars (conducting research in software engineering and maintenance) and three of them are software developers, and have rich experience in handling bug reports. They classify the emotions of the reports independently and then discuss together to share their experiences on why emotion features work in the prioritization of bug reports. Results suggest that 86% of the selected reports having low priority are negative. They are agreed on that being rude when writing a bug report can affect the cohesion of the participants (users or developers). It also affects the prioritization and resolution of bug reports. For example, we observe that *negative* bug reports (e.g., *What is the best way to kill a critical process* or *I am missing a parenthesis but I don't know where*) have *lower* priority not because of *negative* words e.g., *kill* and *missing*, but because they are not constructive (i.e., posting questions but do not present constructive suggestions). In contrast, *positive* bug reports (e.g., *Styled Text printing should implement "print to file"*) have *higher* priority because they often present constructive suggestion and, thus, are likely to be resolved. Another observation is that *adverbs* e.g., *very/too* increase the intensity of the positive/negative emotions and affect the priority for all priorities. Consequently, a respectful environment is feasible and an incentive for new participants.

Moreover, we employ (one-way) ANOVA with the same settings (mentioned in RQ1) to validate the significant

TABLE VI
INFLUENCE OF DIFFERENT INPUT

	CDT			JDT			PDE			Platform		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Default	**67.20%**	**60.44%**	**63.64%**	**63.70%**	**60.52%**	**62.07%**	**74.63%**	**72.34%**	**73.47%**	**59.92%**	**55.58%**	**57.67%**
Disabling Emotion	65.47%	55.86%	60.28%	62.02%	58.43%	60.17%	71.42%	66.80%	69.03%	60.45%	53.57%	56.80%
Disabling Textual Features	65.35%	52.52%	58.24%	61.09%	55.90%	58.38%	71.46%	62.46%	66.66%	61.08%	48.16%	53.86%

difference *cPur* with and without emotion. Results of ANOVA analysis, which presents $F > F_{\text{cric}}$ and p-value $<$ (alpha $=$ 0.05), are not true for F1-score, where $F = 0.39$, $F_{\text{cric}} = 5.98$, and $P_{\text{value}} = 0.55$. We note that in all cases disabling emotion results in a significant reduction in the F1-score. However, the ANOVA analysis suggests that disabling emotion does not significantly influence the F1-score. One possible reason is that the F1-score varies significantly from project to project. For example, it is 57.67% on project Platform whereas it increases dramatically to 73.47% on project PDE. As a result, the variation within groups is even more significant than that between groups. As a result, the ANOVA analysis suggests that there is no significant difference between groups.

Furthermore, we quantify the *effect size* to check the difference of *cPur* with and without emotion by employing Cohen's delta *d*. Result ($d = 0.44$) suggests that the difference of *cPur* with and without emotion is *small*.

Finally, we perform the Pearson correlation coefficient (r) to compute the strength of the relationship between emotion/textual features and priority. Notably, we use the priority prediction of the proposed approach without emotion and without textual features to compute r. Results ($r = 0.405$ and $r = 0.731$, respectively) suggest a *medium* correlation between emotion and priority. Whereas, the correlation between textual features and priority is *large*.

Based on the preceding analysis, we conclude that both textual features and emotion are significantly important for the proposed approach.

G. RQ3: Influence of Preprocessing

The textual information of bug reports contains noisy data (e.g., stop-words and punctuation), which is irrelevant and meaningless (as mentioned in Section III-D). Therefore, passing such information to the machine learning algorithms is an overhead. To this end, applying preprocessing may help in performance improvement and computation cost reduction.

To answer the research question RQ3, we compare the performance results of the proposed approach with and without preprocessing. Evaluation results by enabling and disabling the preprocessing are presented in Table VII. The first column presents the preprocessing input settings. Columns 2–4, 5–7, 8–10, and 11–13 present the performance of project CDT, JDT, PDE, and Platform, respectively. The performance of *cPur* with different settings is presented in the rows of the table. The last row of the table presents the improvement of *cPur* upon different input settings for preprocessing. Fig. 5 also visualizes the performance difference of the proposed approach upon different preprocessing input settings.

From Table VII and Fig. 5, we make the following observations.

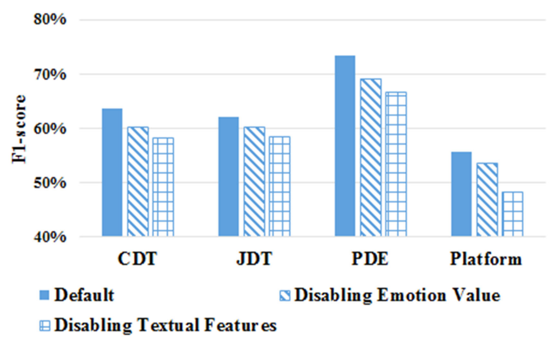


Fig. 4. Influence of different inputs.

1) The proposed approach with the preprocessing step achieves significant improvement in performance. The improvement in precision and recall varies from 1.94% = (67.20%–65.92%)/65.92% to 6.17% = (74.63%–70.29%)/70.29%, and from 5.73% = (60.52%–57.24%)/57.24% to 15.08% = (60.44%–52.52%)/52.52%, respectively.
2) Disabling preprocessing from the input significantly reduces the performance in F1-score of the proposed approach. It varies from 4.17% = (62.07%–59.48%)/62.07% to 9.03% = (73.47%–66.83%)/73.47%.

Moreover, *cPur* and both state-of-the-art approaches (*eApp* and *DRONE*) adopt a preprocessing step to concise the raw text of bug reports. Notably, all approaches use different NLP layers and packages. To compare the results of different layers and packages, we conduct an experiment on the textual data of randomly selected 200 bug reports. All approaches have two common preprocessing steps (*tokenization* and *stop-word removal*) that are in the same sequence. We observe that common preprocessing steps (*tokenization* and *stop-word removal*) produce similar results for *cPur*, *eApp*, and *DRONE*. Both baseline approaches use *Poter stemming algorithms* in their third/final step (*stemming* and *lemmatization*), respectively. However, *cPur* uses *Lancaster stemming algorithm* for *lemmatization* and includes some additional steps as mentioned in Section III-D. We observe that the results of all approaches are different other than the first two steps of preprocessing. The reason for this difference is the selection of the preprocessing tool (e.g., Python NLTK or Stanford Parser) and the parameter settings of the different preprocessing steps (e.g., the use to different stemming algorithms). For example, the output of a word *crying* with *Poter stemming algorithm* is *cri*, which has no meaning in any emotion analysis repository. Whereas, a word *crying* with *Lancaster stemming algorithm* is *cry*, which has negative emotion in emotion analysis. As a conclusion, the selection of preprocessing tools and parameter settings varies from

TABLE VII
INFLUENCE OF PREPROCESSING

	CDT			JDT			PDE			Platform		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Enabled	**67.20%**	**60.44%**	**63.64%**	**63.70%**	**60.52%**	**62.07%**	**74.63%**	**72.34%**	**73.47%**	**59.92%**	**55.58%**	**57.67%**
Disabled	65.92%	52.52%	58.46%	61.90%	57.24%	59.48%	70.29%	63.70%	66.83%	58.57%	48.94%	53.32%

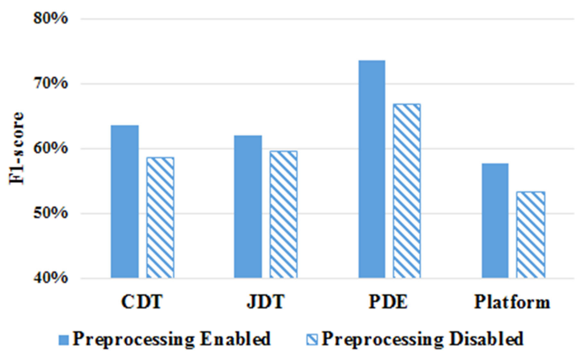


Fig. 5. Influence of preprocessing.

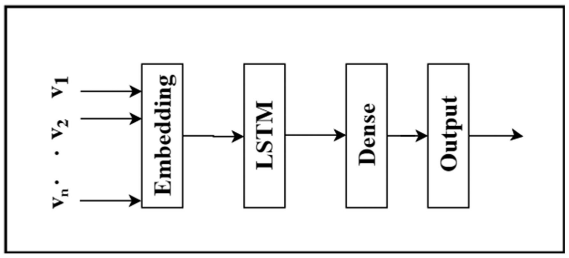


Fig. 6. Overview of LSTM model.

problem to problem as each tool has its own advantages and limitations [38], [51].

Based on the preceding analysis, we conclude that the preprocessing step is significantly important for the proposed approach.

H. RQ4: Influence of Different Lengths of Filters

To answer the research question RQ4, we conduct an additional experiment to choose the filter size of the CNN model. It analyzes how big is the *n* on n-grams that applies a filter to a window of *d* words to generate a new feature. To this end, we first fix all other hyperparameters except filter window size to check the effect of the filter length. Then, we train and test the proposed model with default Adam optimizer for ten epochs on the given dataset.

Results suggest that the average F1-score of the proposed approach with filter size 2, 3, 4, and 5 are 63.09%, 62.95%, 62.81%, and 62.05%, respectively. The CNN with filter windows of size 2 outperforms the other filter windows sizes. It suggests that the meaning of words in a sentence is important and significant in computing the emotion of the sentence. It also proves why bag-of-words (emotion words) can have a strong performance in bug prioritization. In addition, the performance of the CNN model decreases when the filter size increases. However, there is a significant decrease in performance when n-gram reaches to 5. Finally, we combine filters with different window sizes for additional performance improvements and decide to choose the filter sizes of 2, 3, and 4 with 100 feature maps, where the average F1-score is 64.21%.

I. RQ5: Influence of Different Sizes of Training Dataset

To answer the research question RQ5, we conduct an experiment to find the impact of training size on the proposed approach. To this end, we train the proposed model with five different training datasets. We use first 80%, 70%, 60%, 50%, and 40% reports for each training and test the trained models with the last 20% reports. Evaluation results of the proposed approach against each training/testing sample are presented in Table IX.

In Table IX, the first column presents the training sample size. Columns 2–4 present the performance of the proposed approach. The rows of the table present the performance results of the proposed approach against different training sample sizes, respectively.

Results suggest that the average F1-score of the proposed approach with training sizes of first 80%, 70%, 60%, 50%, and 40% reports are 74.95%, 70.24%, 66.52%, 56.19%, and 53.40%, respectively. The CNN with training size 80% outperforms the other training sizes. We observe that the increase in the training size improves the performance of the proposed approach. Although, the *precision* of the proposed approach significantly decreases when the training size is less than 60% (48 000 reports out of 80 000 reports); however, the *recall* does not significantly decrease. Consequently, the results of the proposed approach are acceptable for both *precision* and *recall* when the training size is greater than or equal to 60% of the given dataset.

J. RQ6: Comparison Against Classification Algorithms

To answer the research question RQ6, we apply SVM and LSTM (as mentioned in Section IV-A) for the comparison of their performances with the proposed approach. Note that we use the same preprocessing and emotion features (mentioned in Sections III-E and III-F) to compare these classification algorithms. Moreover, we exploit SVM same as the existing approach (linear SVM with default settings). Whereas, the LSTM model (shown in Fig. 6) contains *embedding layer*, *LSTM layer* (*dropout* = 0.2 and *recurrent_dropout* = 0.2), and *dense layer* (*activation* = *sigmoid*). We use the *binary_crossentropy* as the loss function for *LSTM*.

Evaluation results of classification algorithms are presented in Table VIII. The first column presents the approaches. Columns 2–4, 5–7, 8–10, and 11–13 present the performance of projects CDT, JDT, PDE, and Platform, respectively. Rows 2–4 present the performance of CNN, SVM, and LSTM, respectively. Fig. 7 also visualizes the performance difference in machine learning algorithms.

TABLE VIII
INFLUENCE OF MACHINE LEARNING TECHNIQUES

	CDT			JDT			PDE			Platform		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
CNN	**67.20%**	**60.44%**	**63.64%**	**63.70%**	**60.52%**	**62.07%**	**74.63%**	**72.34%**	**73.47%**	**59.92%**	**55.58%**	**57.67%**
SVM	60.99%	40.69%	48.81%	59.86%	41.24%	48.84%	64.22%	34.79%	45.13%	50.70%	35.83%	41.98%
LSTM	20.11%	44.84%	27.77%	23.90%	48.89%	32.11%	32.80%	57.27%	41.71%	31.49%	56.12%	40.34%

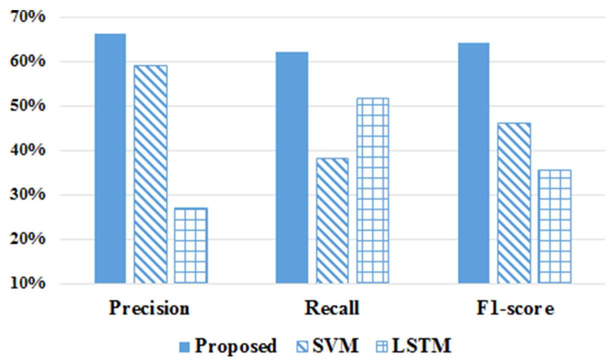


Fig. 7. Influence of machine learning techniques.

TABLE IX
INFLUENCE OF TRAINING SIZE

Training Reports	Precision	Recall	F1-score
First 80%	**76.89%**	**73.11%**	**74.95%**
First 70%	71.63%	68.92%	70.24%
First 60%	67.36%	65.71%	66.52%
First 50%	51.16%	62.32%	56.19%
First 40%	47.97%	60.21%	53.40%

From Table VIII and Fig. 7, we make the following observation.

1) The proposed approach outperforms both machine learning and deep learning algorithms. It achieves the best precision, recall, and F1-score on each of the projects.
2) The proposed approach without preprocessing also outperforms both preprocessing disabled machine learning and deep learning algorithms. The performance improvement of the proposed approach without preprocessing upon SVM (preprocessing enabled) in F1-score varies from 19.77% = (58.46%–48.81%)/48.81% to 48.09% = (66.83%–45.13%)/45.13%. However, the performance improvement of the proposed approach without preprocessing upon LSTM (preprocessing enabled) in F1-score varies from 32.19% = (53.32%–40.34%)/40.34% to 110.52% = (58.46%–27.77%)/27.77%.
3) The performance of LSTM is significantly lower than the proposed approach. The reason for the decrease in performance of LSTM against CNN is that CNN is good at extracting local/position-invariant features, and works well with long input text [12]. However, LSTM performs well with the short and sequential input text. Notably, our input text is long and does not require sequential processing. Thus, CNN works better than LSTM in our case. Another reason for this improvement is the usage of *word2vector* modeling. It captures a large number of precise syntactic and semantic words relationships that could be *n*grams words (verbs having adverb e.g., very good), and assign a value for each word based on their semantic relations as mentioned in Section III-F.
4) The performance of SVM is significantly lower than the proposed approach. The reason for the decrease in performance of SVM upon *cPur* is that SVM works poor with variable-high input dimensions. Whereas, CNN works quite well with variable-high input dimensions. Another reason is that CNN does not need the feature modeling efforts (required for machine learning algorithms), which is tedious and time-consuming.

Based on the preceding analysis, we conclude that the proposed CNN classifier outperforms other machine learning and deep learning classifiers in predicting priority of bug reports.

V. THREATS

A. *Threats to Validity*

A threat to construct validity is the suitability of the selected evaluation metrics. The precision, recall, and F1-score are the standard and most adopted metrics [1], [5], [51]. Therefore, we select these metrics for the evaluation of the classification algorithms.

Another threat to construct validity is the usage of *Senti4SD* for emotion analysis. There are many other libraries for this purpose; however, we select it due to its performance result for the software engineering text. Other repositories for emotion analysis may decrease the performance of the proposed approach.

A threat to internal validity is the implementation of machine learning and deep learning algorithms. To mitigate the threat, we double-check the implementation and results. However, there could be some unseen errors.

A threat to external validity is the generalizability of our results. We evaluate the proposed approach only on the four open-source projects of Eclipse. The inclusion of the bug reports from other projects may decrease the performance of the proposed approach.

Another threat to external validity is the input of the hyperparameters of the deep learning approaches. We trained the deep learning algorithms on a small number of bug reports. They usually require a large training set. They also have a number of hyperparameters to be adjusted. The adjustment of such parameters may influence performance.

VI. CONCLUSION

Bug reports are often submitted either with an incorrect priority level or without defining priority level. Developers read through such bug reports and manually correct or assign the priority of each bug report. Manual prioritization of bug reports requires expertise and resources (e.g., time and professionals). To this end, in this article, we proposed a CNN-based automatic approach for multiclass priority prediction of bug reports. The proposed approach applied not only a deep learning model but also employed natural language techniques and emotion analysis on the given dataset for the priority prediction of bug reports. The proposed approach automated the priority assignment process and saved the required time and efforts of developers. We performed the cross-project evaluation on the history data of the four open-source projects of Eclipse. The evaluation results suggested that the proposed approach outperformed the state-of-the-art approaches.

The broader impact of our article is to show that the textual information of the bug reports could be a rich source of information to prioritize them for their resolution on time. We expect our results to encourage future research on the prioritization of bug. We would like to investigate the rationale behind the proposed approach in future. One of the drawbacks of deep learning neural networks is that it is challenging, if not impossible, to explain why deep learning based approaches, e.g., the one proposed in this article, work or not work. Opening the "black box" of deep neural networks to understand better how a deep learning model learns. Deep learning is called a black box as it is nonparameterized. Although the choice of hyperparameters of deep learning models, such as the number of layers, the activation function, and the learning rate, as well as the predictor importance is known. It is still unclear how machines learn and deduce conclusions. In future, we would like to exploit advanced techniques in neural networks to uncover the rationale behind the phenomenon.

We would also like to investigate a domain-specific prioritization of bug reports by including more bug reports from different domains, e.g., information systems. A domain-specific prioritization of bug reports will affirm the generalizability of the proposed approach.

Acknowledgment

The authors would like to thank the Associate Editor and the anonymous reviewers for their insightful comments and constructive suggestions.

References

[1] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Softw. Eng.*, vol. 20, pp. 1354–1383, Oct. 2015.

[2] Bugzilla, 2018. [Online]. Available: https://www.bugzilla.org/

[3] Jira, 2002. [Online]. Available: https://www.atlassian.com/software/jira/

[4] Github, 2008. [Online]. Available: https://github.com/features/

[5] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Washington, DC, USA, 2013, pp. 200–209.

[6] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange*, New York, NY, USA, 2005, pp. 35–39.

[7] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proc. Softw. Evol. Week—IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng.* , Feb. 2014, pp. 174–183.

[8] M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *Proc. 12th Int. Conf. Mach. Learn. Appl.*, vol. 2, Washington, DC, USA, 2013, pp. 112–116.

[9] Q. Umer, H. Liu, and Y. Sultan, "Emotion based automated priority prediction for bug reports," *IEEE Access*, vol. 6, pp. 35743–35752, 2018.

[10] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *J. Comput. Sci. Technol.*, vol. 27, pp. 397–412, Mar. 2012.

[11] L. Yu, W.-T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," in *Advanced Data Mining and Applications*, L. Cao, J. Zhong, and Y. Feng, Eds., Berlin, Germany: Springer, 2010, pp. 356–367.

[12] B. Wang, "Disconnected recurrent neural networks for text categorization," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers)*, Melbourne, VIC, Australia, Jul. 2018, pp. 2311–2320.

[13] K. Greff, R. K. Srivastava, J. Koutnık, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017.

[14] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46846–46857, 2019.

[15] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Softw. Eng.*, vol. 23, pp. 1352–1382, Jun. 2018.

[16] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories*, May 2010, pp. 1–10.

[17] W. Abdelmoez, M. Kholief, and F. M. Elsalmy, "Bug fix-time prediction model using naïve Bayes classifier," in *Proc. 22nd Int. Conf. Comput. Theory Appl.*, Oct. 2012, pp. 167–172.

[18] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proc. 19th Work. Conf. Reverse Eng.*, Oct. 2012, pp. 215–224.

[19] Y. Tian, N. Ali, D. Lo, and A. E. Hassan, "On the unreliability of bug severity data," *Empirical Softw. Eng.*, vol. 21, pp. 2298–2323, Dec. 2016.

[20] P. Choudhary, "Neural network based bug priority prediction model using text classification techniques," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 5, pp. 1315–1319, 2017.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, vol. 1, pp. 1097–1105.

[22] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2013, pp. 6645–6649.

[23] X. Ouyang, P. Zhou, C. H. Li, and L. Liu, "Sentiment analysis using convolutional neural network," in *Proc. IEEE Int. Conf. Comput. Inf. Technol.; Ubiquitous Comput. Commun.; Dependable, Autonomic Secure Comput.; Pervasive Intell. Comput.*, Oct. 2015, pp. 2359–2364.

[24] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang, "Deep learning in software engineering," 2018.

[25] W.-T. Yih, K. Toutanova, J. C. Platt, and C. Meek, "Learning discriminative projections for text similarity measures," in *Proc. 15th Conf. Comput. Natural Lang. Learn.*, Stroudsburg, PA, USA, Jul. 2011, pp. 247–256.

[26] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," in *Proc. 52nd Annu. Meeting Assoc. Computational Linguistics*, 2014, pp. 655–665.

[27] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, Nov. 2011.

[28] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.

[29] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, May 2010, pp. 45–54.

[30] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, Jun. 2008, pp. 52–61.

[31] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proc. ACM Symp. Appl. Comput.*, 2006, pp. 1767–1772.

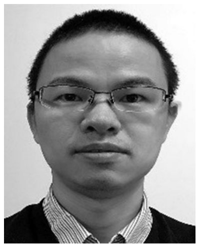
[32] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Jan. 2009, pp. 111–120.

[33] J. Xuan, H. Jiang, H. Zhang, and Z. Ren, "Developer recommendation on bug commenting: A ranking approach for the developer crowd," *Sci. China Inf. Sci.*, vol. 60, Apr. 2017, Art. no. 072105.

[34] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?,” in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, pp. 1–8.
[35] S. Akbarinasaji, B. Caglayan, and A. Bener, “Predicting bug-fixing time: A replication study using an open source software project,” *J. Syst. Softw.*, vol. 136, pp. 173–186, 2018.
[36] P. Bhattacharya and I. Neamtiu, “Bug-fix time prediction models: Can we do better?,” in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 207–210.
[37] Google-Issue-Tracker, [Online]. Available: https://issuetracker.google.com/
[38] E. Loper and S. Bird, “NLTK: The natural language toolkit,” in *Proc. ACL-02 Workshop Effective Tools Methodologies Teaching Natural Lang. Process. Comput. Linguistics*, 2002, vol. 1, 2006, pp. 63–70.
[39] TextBlob, 2013. [Online]. Available: https://textblob.readthedocs.io/en/dev/
[40] J. Uddin, R. Ghazali, M. Mat Deris, R. Naseem, and H. Shah, “A survey on bug prioritization,” *Artif. Intell. Rev.*, vol. 47, pp. 145–180, Apr. 2016.
[41] M. R. Islam and M. F. Zibran, “Sentistrength-SE: Exploiting domain specificity for improved sentiment analysis in software engineering text,” *J. Syst. Softw.*, vol. 145, pp. 125–146, 2018.
[42] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, “Senticr: A customized sentiment analysis tool for code review interactions,” in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 106–111.
[43] F. Calefato, F. Lanubile, and N. Novielli, “Emotxt: A toolkit for emotion recognition from text,” in *Proc. 7th Int. Conf. Affect. Comput. Intell. Interact. Workshops Demos*, 2017, pp. 79–80.
[44] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[45] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” 2012.
[46] T. Young, D. Hazarika, S. Poria, and E. Cambria, “Recent trends in deep learning based natural language processing [review article],” *IEEE Comput. Intell. Mag.*, vol. 13, no. 3, pp. 55–75, Aug. 2018.
[47] Bugzilla, 2018. [Online]. Available: https://bugs.eclipse.org/bugs/
[48] I. Safonov, I. Gartseev, M. Pikhletsky, O. Tishutin, and M. Bailey, “An approach for model assessment for activity recognition,” *Pattern Recognit. Image Anal.*, vol. 25, pp. 263–269, Apr. 2015.
[49] R. E. Schapire and Y. Singer, “Boostexter: A boosting-based system for text categorization,” *Mach. Learn.*, vol. 39, pp. 135–168, May 2000.
[50] E. T. Berkman and S. P. Reise, *A Conceptual Guide to Statistics Using SPSS*. Thousand Oaks, CA, USA: Sage, 2011.
[51] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2008, pp. 346–355.

Qasim Umer received the B.S. degree in computer science from Punjab University, Lahore, Pakistan, in 2006, the M.S. degree in net distributed system development from the University of Hull, Hull, U.K., in 2009, and the second M.S. degree in computer science from the University of Hull, in 2012. He is currently working toward the Ph.D. degree in computer science with the Beijing Institute of Technology, Beijing, China.

He is particularly interested in machine learning, data mining, and software maintenance.



Hui Liu received the B.S. degree in control science from Shandong University, Jinan, China, in 2001, the M.S. degree in computer science from Shanghai University, Shanghai, China, in 2004, and the Ph.D. degree in computer science from Peking University, Bejing, China, in 2008.

He is currently a Professor with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing. He was a Visiting Research Fellow with the Centre for Research on Evolution, Search and Testing (CREST), University College London, London, U.K. He served on the program committees and organizing committees of prestigious conferences, such as International Conference on Software Maintenance and Evolution, RE, International Centre for the Study of Radicalisation, and COMPSAC. He is particularly interested in software refactoring, AI-based software engineering, and software quality. He is also interested in developing practical tools to assist software engineers.

Inam Illahi graduated degree in arts from the University of Sargodha, Sargodha, Pakistan, 2007. He received the M.S. degree in software engineering from the Chalmers University of Technology, Gothenburg, Sweden, in 2010. He is currently working toward the Ph.D. degree in software engineering with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China.

He is particularly interested in software maintenance, crowdsourcing, and machine learning.