Noname manuscript No.
(will be inserted by the editor)

On the Classification of Bug Reports to Improve Bug Localization

Fan Fang · John Wu · Yanyan Li · Xin Ye · Wajdi Aljedaani · Mohamed Wiem Mkaouer*

Received: date / Accepted: date

Abstract Bug Localization is the automated process of finding the possible faulty files in a software project. Bug localization allows developers to concentrate on vital files. Information retrieval (IR)-based approaches have been proposed to assist automatically identify software defects by using bug report information. However, some bug reports that are not semantically related to the relevant code are not helpful to IR-based systems. Running an IR-based reporting system can lead to false-positive results. In this paper, we propose a classification model for classifying a bug report as either uninformative or informative. Our approach helps to lower false positives and increase ranking performances by filtering uninformative information before running an IR based bug location system. The model is based on implicit features learned from bug reports that use neural networks and explicit features defined manually. We test our proposed model on three open-source software projects that contain over 9,000 bug reports. The results of the evaluation show that our model enhances the efficiency of a developed IR-based system in the trade-off between precision and recall. For implicit features, our tests with comparisons show that the LSTM-network performs better than the CNN and multi-layer perceptron with respect to the F-measurements. Combining both implicit and explicit features outperforms using only implicit features. Our classification model helps improve precision in bug localization tasks when precision is considered more important than recall.

Keywords Bug Classification · Bug Localization · Bug Report Quality · Machine Learning

Fan Fang
California State University
E-mail: fang014@cougars.csusm.edu

John Wu
California State University
E-mail: wu028@cougars.csusm.edu

Yanyan Li
California State University
E-mail: yali@csusm.edu

Xin Ye
California State University
E-mail: xye@csusm.edu

Wajdi Aljedaani
University of North Texas
E-mail: wajdialjedaani@my.unt.edu

Mohamed Wiem Mkaouer
Rochester Institute of Technology
E-mail: mwmvse@rit.edu

1 Introduction

A software *bug report* is a descriptive document used to record the scenario of a software product's unexpected behaviors. It represents the backbone of program repair, as it allows developers to be aware of all software anomalies, either caught in the testing process, or experienced by end users. Such anomalies are known as *bugs* or *defects* (Bruegge and Dutoit, 2009). There defects undergo a life cycle that starts with its triage for verify whether it is a known bug that has been recently reported. If it is not a duplicate bug, then it undergoes the assignment, which is the process of finding the most suitable developer to debug it. Once a developer is assigned, the assignee analyzes the report description and tries to reproduce the problem in order to locate the faulty file(s) (LaToza and Myers, 2010). This process is known as bug localization (Bacchelli and Bird, 2013). Once a patch is generated, the patched code is re-tested to validate the disappearance of the bug. This process is known to be manual intensive, which can

negatively impact team productivity especially that the number of bug reports tends to be proportional to software growth. For instance, the Eclipse Platform project team received 1,567 bug reports in 2017 alone[1], averaging four bug reports a day. Thus, the preciseness of the information in the bug report is important to facilitate the discovery of the bug (Buse and Zimmermann, 2012). However, due to the various possible sources reporting bugs, it is hard to guarantee that all bug reports contain sufficient information to easily debug them, along with the explosion of the number of tightly-coupled classes and modules to investigate, makes the bug-fixing process nontrivial and challenging (Breu et al, 2010).

As the investigation of these reports is tedious and time-consuming (Murphy-Hill et al, 2013), various studies have been proposed to automate this process by preselecting candidate files that are most likely to contain the bug. These studies rely on creating a similarity between the reported bug text information, and the corresponding software artifacts. Since bug reports, and software artifacts are written in either natural language or source code, these approaches heavily rely on Information Retrieval (IR)to define their similarity function. In order to develop this similarity function, intelligent algorithms have been deployed to learn from previously solved bug reports by creating features that capture the closeness of the bug report and its corresponding infected file(s). In this context, recent studies have been relying on Machine Learning (ML) and Deep Learning (DL) to localize files for bug reports (Lam et al, 2015; Huo and Li, 2017; Huo et al, 2016). Similaryly, we developed, in our previous studies (Ye et al, 2014, 2016), a learning-to-rank model combining hand-defined *features* to rank top potential faulty files. Many other ML and DL-based approaches, deploying a variety of IR-based techniques to calculate features were also investigated in the literature, such as the Vector Space Model (VSM) (Zhou et al, 2012; Saha et al, 2013), Naïve Bayes (Kim et al, 2013), Latent Dirichlet Allocation (LDA) (Lukins et al, 2010; Nguyen et al, 2011), combinations of VSM and LDA (Rao and Kak, 2011), etc.

These IR-based approaches, unlike some other spectrum-based approaches (Cleve and Zeller, 2005; Dit et al, 2013; Poshyvanyk et al, 2013, 2007; Liu et al, 2005; Jin and Orso, 2013; B. Le et al, 2016; Le et al, 2015; Jones and Harrold, 2005) that use runtime execution information to locate bugs, do not require running test cases. However, because they rely on the bug report content, the uneven quality of bug reports can be an impediment to their performance.

According to a user study by Bettenburg et al (2008), in which they receive responses from 446 developers, a mismatch is discovered between whatever is considered by developers as useful and what bug reports are describing. They consider reports to be of good quality, if they easily facilitate the replication of a given bug. However, not all bug reports are of similar quality. For instance, incomplete or even ambiguous information can be found in bug reports (Bettenburg et al, 2008; Kim et al, 2013; Hooimeijer and Weimer, 2007).

Furthermore, there are reports that may be useful to programmers but not necessarily helpful for IR-based techniques. For instance, if we consider bug 305571[2] from the Eclipse platform, it describes a given problem as: "*Links in forms editors keep getting bolder and bolder*". When reading this reports, developers were able to investigate the files related to menus and figures, through group discussion that the buggy file was *TextHyperlinkSegment.java*. However, *TextHyperlinkSegment.java* has no explicit semantic relationship with the bug report. So, using the VSM of Lucene[3] for the purpose of ranking all candidate files, none of the top ranked ones were relevant. VSM has been recommending files that are semantically closer to the given description, and so, files related to the forms were making the top like *FormPage.java* and *FormEditor.java*.

So, running IR-based techniques on such bug reports, being semantically far from existing source code, would lower the precision of these techniques. Therefore, if we can identify them early in the bug localization process, then keeping silent can decrease the false positiveness of the model.

In this context, Kim et al (2013) proposed a two-phase model that first classifies bug reports into either "informative" or "deficient" and then locates bugs for only "informative" reports. Their model uses fixed buggy files as labels and applies Naïve Bayes to classify an "informative" report to a specific label (buggy file). However, if a new buggy file has not been fixed before, it would not be considered as a label and hence can not not be located. To address this bug relevancy problem, Kim's work inspired us to, before applying a specific IR-based system to find the bug, perform a bug report classification to filter out deficient reports and reports that are unhelpful to the IR-based system. By filtering out deficient reports, we believe that our bug localization will be more accurate at the expense of recommending solutions for a lesser number of reports.

Our work deploys the Long Short-Term Memory (LSTM) model to create a filtering process that is initiated before processing any bug reports for localization. Our model gets as input a open bug report, and outputs a binary decision on whether this report is "informative"

[1] https://bugs.eclipse.org/bugs/

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=305571

[3] https://lucene.apache.org/core/2_9_4/scoring.html

or "uninformative" with regard to any IR-based bug localizer. In a nutshell, our model is a Recurrent Neural Network (RNN) with LSTM units (Hochreiter and Schmidhuber, 1997) for learning features from sequence data. LSTM has been recently investigated for the resolution of Software Engineering problems (Choetkiertikul et al, 2018; Huo and Li, 2017). We use LSTM to learn from bug reports their vector representations, which then serve as input *features* to a *Softmax* layer for classification. Once the model is trained, it would be able to classify a given report as "informative", or being safe to be used by IR-based approaches, otherwise, if it classifies the report as "uninformative", then the IR-based localization is skipped.

To test the efficiency of our classification, we tested our LSTM binary classifier using 9,000 bug reports, extracted from three large-scale open-source Java projects, widely used in previous bug localization studies. The validation shows that, under a trade-off between the classification recall and precision, the LSTM binary classifier is able to support IR-based localizers in increasing their accuracy and obtain more precise ranking results.

Our experiments contain a comparative study between LSTM and Convolutional Neural Network (CNN) (Lecun et al, 1998) (another class of DNN that is recently used to solve SE tasks (Le et al, 2015; Xu et al, 2016; Mou et al, 2016)), multilayer perceptron (Hornik et al, 1989), and a simple baseline approach classifying a bug report based on its length under the assumption that larger descriptions are most likely to contain more relevant content. Our findings demonstrates that our baseline is as competitive as the the multilayer perceptron, while being outperformed by both LSTM and CNN. However, our model strikes the best trade-off between precision and recall.

The remainder of this paper is structured as follows. Section 2 provides an overview of the bug locating of the pre-filter approach. Section 3 explains the bug report classification adaptation of the LSTM network. In Section 4, we present CNN classification. In Section 5, a multi-layer perceptron is discussed. Section 6 displays our explicit features for bug report classification. Section 7 illustrates the setup and results of the evaluation. We discribe our threats to validity in Section 8. Following a discussion of related work in Section 9, the paper ends in Section 10 with future work and concluding remark.

2 High Level Architecture of Bug Report Pre-Filtering

Our approach overview is outlined in Figure 1. Our pre-filtering approach is activated whenever a verified bug report is received. It preprocesses its textual information, and feeds it into the neural network, already trained using previous reports and their corresponding source code. The model makes the binary decision of either classifying the bug report as "informative" and "uninformative". A bug report is considered *informative* if its corresponding textual information is similar to those of reports whose infected files were successfully identified by the IR-based localizers. In that case, the bug reports can be used as input to the localizer in place, which leverages the report content to rank top source code files that are most likely related to the given bug report. On the other hand, a bug report with the label "uninformative", is considered unhelpful to the IR-based localizer. Note that an uninformative report can be also given to the IR-based localizer, but the chances of the loclizer's success in finding the adequate source files to recommend is unknown, and eventually lower than the chances of an informative one, as we will later demonstrate in the experiments. Note that we did not specify any localizer, since there are many existing techniques that are compatible with our proposed approach.

3 Bug Report Classification using an LSTM Network

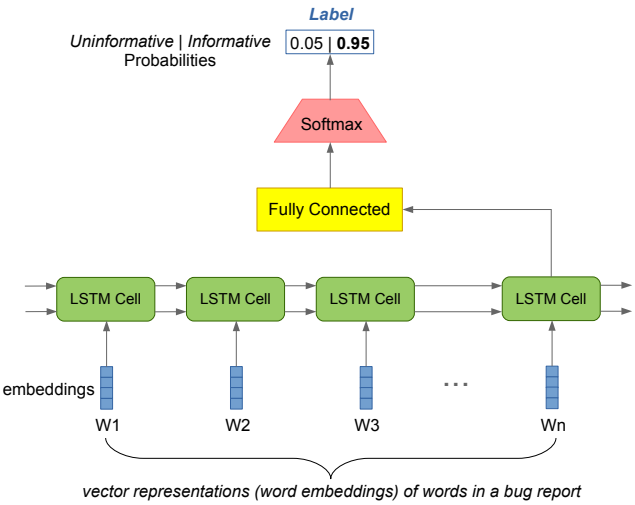

Fig. 2: Bug-report classification architecture: using LSTM.

Our classification model's architecture is sketched in Figure 2. For an input report, it transforms its works into their vector representation, which are then sent as input to a Recurrent Neural Network (RNN) implemented with LSTM units. The output of the LSTM

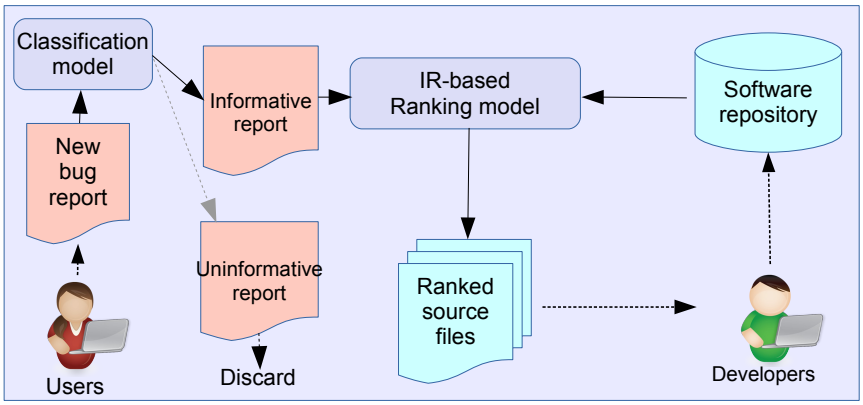

Fig. 1: High level architecture: pre-filtering before ranking.

unit at the last time step is fed into a fully connected layer, followed by the Softmax model that produces the categorical distribution.

In the coming subsections, we provide details of each step.

3.1 From Bug Report to Bug Report Matrix

In a list of bug reports, we combined the description and summary of the bug report into a single document. We applied the data pre-processing on the document where any numerical number and text punctuation will be removed. Next, we separated the text by white space to acquire a bag-of-words$T$ of the document: $T = (w_1, w_2, w_3, ...w_N)$, where the word token in the study is w_i, and the total number of words is N.

Then we will display each word-token wi with a d-dimensional vector of an actual-number $\mathbf{w}i$, which names word-embedded capturing those contextual semantics meaning Levy and Goldberg (2014). We utilize the Mikolov's Skip-gram Le and Mikolov (2014) to the study in order to learn the model of the word embedding with the size of $d1$ on the dumps of Wikipedia data[4], the reference of the programming language as well as project development documents.

Bug reports may have different lengths. RNN can work on variable-length sequence input. However, when we train and update the LSTM network, we use multiple bug reports (e.g., 64) in a mini-batch to compute the gradient of the cost function at each step. For simplicity, we can set a fixed size of $d2$ to all the bug reports so that a training batch can be represented by a single tensor in the TensorFlow implementation of RNN[5]. A bug report with less than $d2$ words will be padded with zero vectors.

[4] https://dumps.wikimedia.org/enwiki/
[5] https://www.tensorflow.org/tutorials/recurrent

Then the original bag-of-words T of a bug report can be converted into a matrix of real numbers: $\mathcal{M} \in \mathbf{R}^{d1xd2}$, where $\mathcal{M} = (\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, ..., \mathbf{w}_{d2})$ and $\mathbf{w}_i \in \mathbf{R}^{d1}$ is the embedding of word w_i. We call this matrix a bug report matrix that serves as input to the LSTM network.

3.2 From Bug Report Matrix to Feature Vector

An LSTM network is a RNN using LSTM units in the hidden layer, where an LSTM unit is composed of a *memory cell* and three multiplicative gates (an *input gate*, an *output gate*, and a *forget gate*) Hochreiter and Schmidhuber (1997). Traditional RNNs are difficult to train on long sequences due to the vanishing gradient problem Bengio et al (1994). LSTM networks effectively alleviate this problem by using the multiplicative gates to learn long-term dependencies over long periods of time.

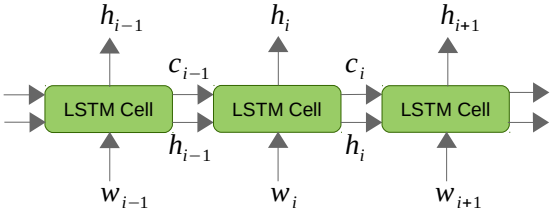

Fig. 3: An LSTM Network.

The LSTM network takes the bug report matrix $\mathcal{M}$ as a time series input (from $\mathbf{w}_1$ to $\mathbf{w}_n$). At each time step, as shown Figure 2, an embedding $\mathbf{w}_i$ is fed into the LSTM network, where the output $\mathbf{h}_i \in \mathbf{R}^m$ of an LSTM unit is determined based on three types of input: the current embedding $\mathbf{w}_i$, the previous LSTM output $\mathbf{h}_{i-1}$, and the content of the memory cell $\mathbf{c}_{i-1} \in \mathbf{R}^m$

from the previous time step, where m is the number of hidden units (states) in the memory cell.

The output of the LSTM unit from the last time step $\mathbf{h}_n$ will be used as the final output $\mathbf{h}$ of the LSTM network. It is a feature vector representation of the original bug report that captures the structural and semantic dependencies.

3.3 From Feature Vector to Categorical Distribution

The output of the LSTM network $\mathbf{h} \in \mathbf{R}^m$ is fed into a fully connected layer with rectifier (ReLU) Nair and Hinton (2010) activation function.

$$\mathbf{x} = max(\mathbf{f}, 0), \qquad \mathbf{f} = \mathbf{U}^T\mathbf{h} + \mathbf{b} \tag{1}$$

The output $\mathbf{x} \in \mathbf{R}^n$ of the fully connected layer is shown in Equation 1, where $\mathbf{U} \in \mathbf{R}^{mxn}$ is the weighting matrix initialized using the Glorot uniform scheme Glorot and Bengio (2010), $\mathbf{b} \in \mathbf{R}^n$ is the bias, and $n = 2$ is the number of categories. It serves as input to a Softmax model.

Softmax normalizes $\mathbf{x} \in \mathbf{R}^n$ into a new n-dimensional vector $\tilde{\mathbf{y}}$ with real numbers in the range $[0, 1]$. The elements of $\tilde{\mathbf{y}}$ sum up to 1. So it can be used as the categorical (probability) distribution over all the possible categories: "informative" and "uninformative".

$$\tilde{y_i} = P(r \in i|\mathbf{x}) = \sigma(\mathbf{v}_i^T\mathbf{x}) = \frac{exp({\mathbf{v}_i}^T\mathbf{x})}{\sum_{k=1}^{2} exp({\mathbf{v}_k}^T\mathbf{x})}, i \in [1, 2] \tag{2}$$

Given $\mathbf{x}$, the probability of the i^{th} category for bug report r is denoted in Equation 2, where $\mathbf{v}$ is the weighting vector.

Finally, the bug report is classified into the category with the largest probability value.

3.4 Model Training

Parameters of the LSTM network, fully connected layer, and the Softmax model are trained on minimizing the cross-entropy error using Adam (adaptive moment estimation) optimizer Kingma and Ba (2014).

$$J(w) = \sum_{r \in R} \sum_{i=1}^{n} (y_i \log \tilde{y_i} + (1 - y_i) \log(1 - \tilde{y_i})) \tag{3}$$

The cross-entropy cost function is shown in Equation 3, where y_i is the observed probability of category i for bug report r, $\tilde{y_i}$ is the estimated probability, R denotes a training batch.

Before training, the training set is split into small *batches*. During training, the models are updated using the gradient of the cost function computed over a mini-batch set. Using batches improves training efficiency, helps avoid local minima, and achieves better convergence Goodfellow et al (2016).

One cycle (a forward pass and a backward pass) of seeing all the training data is called an *epoch*. Let T denotes the training set and R be a mini-batch, the number of batches is $num\_batches = |T||R|^{-1}$. So each epoch updates the models $num\_batches$ times.

The models are trained over a maximum 500 epochs with an earlier stopping criterion, which deems convergence when seeing a certain number (e.g., 10) of continuous performance degrade on the validation dataset.

To achieve more robust convergence, we also apply the *variational dropout* technique Gal and Ghahramani (2016) during training, which reduces over-fittings by randomly cleaning up some input, output, and hidden units.

4 CNN for Bug Report Classification

A CNN is a deep feedforward neural network composed of one or more convolutional layers with subsampling (poolings) Lecun et al (1998). Unlike RNNS that memorize the past and use the previous output to update the current states, information in CNNs passes through in one direction and never go back. While LSTM networks are robust for learning long-term dependencies from time series, CNNs learn dependencies from the spatial locality and work effectively on 2-D structure (e.g., image) Russakovsky et al (2014).

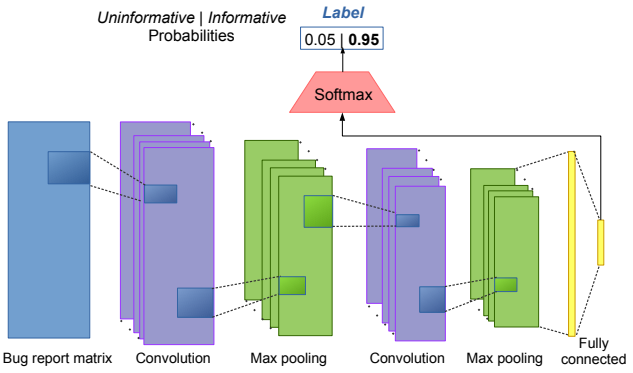

Fig. 4: Bug-report classification architecture: using CNN.

Figure 4 shows the overall architecture of using CNN for bug report classification. In this paper, we use a CNN with two convolutional layers with max poolings followed by two fully connected layers with *sigmoid* the activation function. It takes as input a bug report matrix $\mathcal{M} \in \mathbf{R}^{100x100}$ as described in Section 3.1. The first convolutional layer uses eight fixed-size (5x5) filters to perform convolution operations over the input matrix and outputs the same number of *feature maps*. A max-pooling layer reduces the size of the feature maps by subsampling. The second convolutional layer using sixteen filters takes as input the output of the first layer. The fully connected layers (with fan-out of 512 and 2 respectively) project the sixteen feature maps from the second convolutional layer to a vector that serves as input to the Softmax model for computing the probability distribution.

We use the same training procedure as discussed in Section 3.4 to train the models (CNN and Softmax) by minimizing the cross-entropy cost function per mini-batch over a maximum of 500 epochs with an early stopping criterion.

5 Multilayer Perceptron for Bug Report Classification

A multilayer perceptron is a feedforward neural network that projects data from the input layer to a linear separable space through multiple hidden layers with activation functions (Hornik et al, 1989).

$$f_1(\mathbf{x}) = G_1(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1),\ f_2(\mathbf{x}) = G_2(\mathbf{W}_2 f_1(\mathbf{x})+\mathbf{b}_2),\ ... \tag{4}$$

Given input $\mathbf{x}$, the output $f_i(\mathbf{x})$ of the i^{th} hidden layer is shown in Equation 4, where $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weights matrix and bias, G_i is the activation function.

We use a multilayer perceptron with three hidden layers, all using 500 computation nodes and *sigmoid* the activation function. A bug report matrix $\mathcal{M}$ is unpacked to a vector that serves as input to the first hidden layer. The output of the last hidden layer is fed into a fully connected layer, followed by a softmax model to estimate the categorical probabilities.

The models are trained by minimizing the cross-entropy function using the same training scheme, as discussed in the previous sections.

6 Explicit Features for Bug Report Classification

Besides using neural networks to learn implicit features, we design explicit features for bug report classification. Table 1 shows the design of explicit features, where features about action items are from a related work (Bettenburg et al, 2008). When a bug report is received, it will be represented as a normalized feature vector of all the explicit and implicit features. The feature vector will serve as input to supervised learning models such as support vector machine (SVM) and k-nearest neighbor for classification. The learning models will be trained using a dedicated training set.

$$feature\ 1 = \mid r.summary \mid \tag{5}$$

Feature 1 measures the size of the report summary, where r refers to the bug report, and $r.summary$ refers to the report summary represented as a bag of words.

$$feature\ 2 = \mid r.description \mid \tag{6}$$

Feature 2 measures the size of the report description, where $r.description$ refers to a bag of words of the report description.

$$feature\ 3 = \begin{cases} 1 & if\ r\ contains\ code\ samples \\ 0 & otherwise \end{cases} \tag{7}$$

Feature 3 checks if the bug report r contains code samples. We implemented a parser based on related work (Moonen, 2001) to identify Java code samples. If the report r contains code samples, we set feature 3 to be 1. Otherwise, it will be 0.

$$feature\ 4 = \begin{cases} 1 & if\ r\ contains\ attachments \\ 0 & otherwise \end{cases} \tag{8}$$

Feature 4 checks if the bug report r contains attachments. We use regular expression (Bettenburg et al, 2008) to identify attachments. We do not distinguish the types of attachments. We consider different types of attachments (e.g., patches, screenshots) are all important to provide helpful information. If the report r contains attachments, we set the feature to be 1 and 0 otherwise.

$$feature5 = \begin{cases} 1 & if\ r.summary\ contains\ class\ names \\ 0 & otherwise \end{cases} \tag{9}$$

Table 1: Explicit and implicit features.

Structural features:
$feature1$ – the length of the report summary
$feature2$ – the length of the report description
$feature3$ – if the report contains code samples or not
$feature4$ – if the report contains attachments or not
Keyword-related features:
$feature5$ – if the report summary contains class names or not
$feature6$ – if the report descriptions contains class names or not
$feature7$ – if the report summary contains method names or not
$feature8$ – if the report descriptions contains method names or not
$feature9$ – the number of action items (e.g., open, select, click) Bettenburg et al (2008)
Neural Network features:
$feature10$ – LSTM categorical probabilities
$feature11$ – CNN categorical probabilities
$feature12$ – multilayer perceptron categorical probabilities

If a bug report mentions a class name in its summary, that class is very likely to be relevant to the bug. The appearance of class names in the report summary is a signal that the report may be informative. For a report r, if its summary $r.summary$ contains class names, we set feature 5 to be 1 and set to 0 otherwise.

$$feature6 = \begin{cases} 1 & if\ r.description\ contains\ class\ names \\ 0 & otherwise \end{cases} \tag{10}$$

Feature 6 checks if the description $r.description$ of a bug report r contains class names. If yes, we set it to 1. If not, we set it to 0.

$$feature7 = \begin{cases} 1 & if\ r.summary\ contains\ method\ names \\ 0 & otherwise \end{cases} \tag{11}$$

The appearance of a method name in the summary $r.summary$ of a bug report r can also be a signal that the method is relevant to the bug. Feature 7 checks if the summary contains method names or not.

$$feature8 = \begin{cases} 1 & if\ r.description\ contains\ method\ names \\ 0 & otherwise \end{cases} \tag{12}$$

Similarly, feature 8 is designed to check if the description $r.description$ of a bug report r contains method names or not.

$$feature\ 9 = |\ action\ items\ | \tag{13}$$

According to a related work (Bettenburg et al, 2008), action items (keywords about actions, e.g., open, select, click) could be a signal indicating that the report may provide helpful information to reproduce and locate the bug. We follow this work to capture action items in the report content, including summary and description. Given a bug report, feature nine is designed to measure the total number of action items in the report.

$$feature\ 10 = Prob_{lstm}(\text{"}informative\text{"}\ |\ r) \tag{14}$$

Feature 10 is the LSTM-network output categorical probabilities as discussed in Section 3.3. More specifically, $Prob_{lstm}(\text{"}informative\text{"}\ |\ r)$ refers to the probability of the given bug report r being classified as "informative".

$$feature\ 11 = Prob_{cnn}(\text{"}informative\text{"}\ |\ r) \tag{15}$$

Feature 11 is the output of the CNN model as discussed in Section 4, where $Prob_{cnn}(\text{"}informative\text{"}\ |r)$ refers to the probability of the given bug report r being classified as "informative".

$$feature\ 12 = Prob_{mlp}(\text{"}informative\text{"}\ |\ r) \tag{16}$$

Feature 12 is the multilayer perceptron output as discussed in Section 5, where $Prob_{mlp}(\text{"}informative\text{"}\ |\ r)$ refers to the probability of the given bug report r being classified as "informative".

6.1 Feature Scaling

Feature scaling helps normalize different types of features within the same range between 0 and 1 so that they are comparable with each other. Since many features in Table 1 are already within the range between

0 and 1, we only perform feature scaling for features 1, 2, and 9.

$$\phi' = \begin{cases} \frac{\phi - \phi.min}{\phi.max - \phi.min} \end{cases} \quad (17)$$

Given the original feature ϕ, let $\phi.min$ and $\phi.max$ be the minimum and the maximum observed values, ϕ' is the normalized feature.

6.2 Bug Report Classifications using All Features

When a bug report is received, we will calculate its features in Table 1. The report will be represented as a vector of twelve features. The feature vector will serve as input to a classifier, which will classify the report as either "informative" or "uninformative". In this study, we compared four classifiers, including Support Vector Machine (SVM), K-Nearest Neighbours (KNN), Decision Tree, and Naive Bayes.

7 Evaluation

The goal of our experiments is to challenges the performance and feasibility of our classifier, as part of the larger bug localization process. We first want to see how accurate is our classification, in terms of the distinction between relevant and non-relevant reports with regard to IR-based localizers. Also, our ultimate goal is to improve the accuracy of these localizers with the presence of our tool. We would also like to see the difference in output between the Section 3, the Section 4, the Section 5 and section 6 using multiple classifiers.

7.1 IR-based localizers

Our approach does not require a specific localizer to work with. For our expeirment, we deploy our own recent localizer Ye et al (2016). We choose to use as 1) it is available; 2) its localization is IR-Based; 3) it is competitive with other state of the art localizers Lam et al (2015, 2017). Our localizer deploys a ***L**earning-to-**R**ank* model that gives weights to **W**ord-**E**mbedding-based features and VSM-based features to rank corresponding files for a given bug report. In this study, we call it *LRWE*.

7.2 Benchmark Datasets

We use our own dataset, proposed in our initial bug localization investigation (Ye et al, 2014). This dataset contains 9,105 bug reports, extracted from famous open source projects, namely Eclipse Platform UI, JDT, and SWT. each bug report contains a textual description of the anomaly, its reporter, the ID of the developer who wrote the fix, along with, most importantly, the set of files that were fixed as part of this bug resolution. Such dataset has been used as the ground truth for many bug localization and bug assignment studies (Ye et al, 2014, 2016; Lam et al, 2015, 2017; Dilshener et al, 2016). Those bug reports were categorized into three classification model. The first category is the training dataset to train the classification model based on 5,000 bug reports. The second category is the testing dataset, where is the classification test on 2,596 bug reports. The last category is the validation dataset, where we validate our classification model on 1,500 bug reports of whether it is covered or not.

7.3 Labeling the Data

The main aim of our classification experiment is to filter out the bug report that is not useful while maintaining the desired bug report to the IR-Based system for increasing the locating performance and decreasing the false positives. Therefore, labeling the data is the basis of our experiment. Due to utilizing an automatic and reliable process for assessing our experiment, the IR-Based system used to determine the usefulness of bug reports in either "informative" or "uninformative".

Due to utilizing an automatic and reliable process for assessing our experiment, the IR-Based system used to determine the usefulness of bug reports in either "informative" or "uninformative". More precisely, we execute the bug locating system LRWE on every individual bug report that we have on the corresponding source files that examined from commit rights of projects before the bug fixing commits. The system displays the output of all the entire source code files. The list of the source code files is sorted based on the bug report label as "predictable" if there is at least one buggy file that places in the top N positions. Otherwise, if the source code files ranked in the list of top N positions, these files are not relevant, and we label them as "uninformative" for the bug report.

According to Miller's 7 ± 2 law (Miller, 1956), individuals are able to perform concurrently seven more or less two activities. So we select $N = 10$if there is a real bug located in the top 10 suggestions, supposing a list is helpful to users.After we performed the data labeling process, all of our bug reports in the dataset, which are 9,105, should be labeled as either "informative" or "uninformative".

Table 2: Benchmark Projects: bug reports are split into a testing, a validation and a training set.

Project	Time Range	# reports for testing	# reports for validation	# reports for training	total
Eclipse Platform UI	2001-10 – 2014-01	1,156	500	2,000	3,656
SWT	2002-02 – 2014-01	817	500	1,500	2,817
JDT	2001-10 – 2014-01	632	500	1,500	2,632

7.4 Evaluation Metrics

From two points of view, we examine our classification of bug reports. We would like first to test the efficiency of its classification. Second, we would like to check if it will help to increase the performance of the IR-based bug localization system. We then perform studies using the following evaluation measurements:

- ***True Positive***: an "informative" bug report that is also classified as "informative"
- ***False Positive***: an "uninformative" bug report that is classified as "informative"
- ***True Negative***: an "uninformative" bug report that is also classified as "uninformative"
- ***False Negative***: an "informative" bug report that is classified as "uninformative"
- ***Accuracy***: a standard metric measuring the correctness of the classification results.

$$Accuracy = \frac{|true\ positives| + |true\ nagatives|}{the\ totaly\ number\ of\ instances} \tag{18}$$

- ***Precision***: a standard metric measuring the usefulness of the classification results.

$$Precision = \frac{|true\ positives|}{|true\ positives| + |false\ positives|} \tag{19}$$

- ***Recall***: a standard metric measuring the completeness of the classification results.

$$Recall = \frac{|true\ positives|}{|true\ positives| + |false\ negatives|} \tag{20}$$

- ***F-Measure***: a standard metric combining both Precision and Recall to measure the classification performance.

$$F - Measure = (1 + \beta) \cdot \frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{21}$$

When we give equal weights to Precision and Recall by setting $\beta = 1$, we have **F1-Measure** that is called the harmonic mean of Precision and Recall.

When we give more weights to Precision by setting $\beta = 0.5$, we have **F0.5-Measure** that considers Precision is more important.

- ***Mean Average Precision (MAP)***: a standard metric measuring the overall ranking performance of an IR system (Manning et al, 2008). It is defined as the mean of the average precision overall queries. MAP is widely used in measuring the ranking performance of IR-based bug locating systems (Huo and Li, 2017; Huo et al, 2016; Lam et al, 2015; Saha et al, 2013; Ye et al, 2014, 2016; Zhou et al, 2012).
- ***Mean Reciprocal Rank (MRR)***: a metric measuring the ranking performance of an IR system on the first recommendation (Voorhees, 1999).

In order to determine the effects of the bug report classification, we utilized Accuracy, Precision, Recall, F1-Measure, and F0.5-Measure. To calculate the effects of the IR method, we utilized MAP and MRR.

7.5 Training the Neural Network Models

```
max_epochs = 500
patience = 10
best_accuracy = 0
prior_accuracy = 0
for(epoch = 0; epoch < max_epoch; epoch++):
    train the model over all the batches in one cycle
    run the model on the validation set
    if Accuracy > best_accuracy:
        best_accuracy = Accuracy
        test the model on the testing set
        patience = 10
    else if Accuracy > prior_accuracy:
        patience = 10
    else:
        patience = patience - 1
        if patience < 0:
            return the testing results
    prior_accuracy = Accuracy
test the model on the testing set
return the testing results
```

Fig. 5: The training and testing procedure.

Table 3: Classification results of using difference models: MLP refers to the multilayer perceptron model, and NC (No Classification) refers to classifying all the instances (bug reports) as positive ("informative").

Project	Metric	LSTM	CNN	MLP	NC
Eclipse Platform UI	Accuracy	**0.670**	0.650	0.645	0.658
	Precision	**0.703**	0.672	0.676	0.658
	Recall	0.862	0.901	0.884	**1**
	F1-Measure	0.775	0.770	0.766	**0.794**
	F0.5-Measure	**0.730**	0.708	0.709	0.706
SWT	Accuracy	**0.694**	0.685	0.692	0.692
	Precision	0.694	0.698	0.692	0.692
	Recall	**1**	0.963	0.884	**1**
	F1-Measure	**0.819**	0.809	0.783	0.818
	F0.5-Measure	**0.739**	0.738	0.738	0.738
JDT	Accuracy	**0.763**	0.747	0.710	0.759
	Precision	**0.762**	0.761	0.758	0.759
	Recall	**1**	0.971	0.908	**1**
	F1-Measure	**0.865**	0.853	0.823	**0.863**
	F0.5-Measure	**0.8**	0.796	0.784	0.797

As discussed in Section 3.4, the neural network models (LSTM-network, CNN, and multilayer perceptron) are trained over a maximum of 500 epochs with an early stopping criterion.

The descriptions of the training and testing process are shown in figure 5. Once the models are trained in every era, we check and monitor the performance of the validation dataset models. When the validation accuracy is steadily decreased over Ten times, we consider that the models are converged, and the training phase ends. Finally, we finalize the testing results if the models achieve the optimal validation accuracy. In Table 1, present the testing results based on the testing dataset, which are utilized as features 10, 11, and 12.

7.6 Tuning the Hyperparameters

We tune models' hyperparameters on the validation dataset using the same procedure as shown in Figure 5 but without testing on the testing set.

- LSTM-network, the number of hidden units in the memory cell is set to 32, the dropout rate on the input layer is 0.9, the output dropout rate is 0.7, and the learning rate 0.003.
- CNN, the first convolutional layer uses eight filters with size 5x5. The second convolutional layer uses sixteen 5x5 filters. The pooling size is 2x2. The fan-out of the first fully connected layer is 512.
- Multilayer perceptron, all three hidden layers use 500 internal nodes.
- The learning rates for both CNN and perceptron are 0.001. The training batch size for these three models are all set to 64.
- SVM, KNN, Decision Tree, and Naive Bayes, we train these classifiers on the training dataset and tune their hyperparameters on the validation dataset as well.

7.7 Results and Analysis

The remainder of this section describes the findings of the study and answering our four research questions.

> ***RQ1:*** Can using only the neural network models help filter out "uninformative" bug reports?
> ***RQ2:*** What is the difference between using different neural network models?
> ***RQ3:*** Can using both the explicit features and the implicit features (neural-network features) help filter out "uninformative" bug reports?
> ***RQ4:*** What is the performance of our proposed classification approach in the task of bug localization?

7.7.1 ***RQ1: Can using only the neural network models help filter out "uninformative" bug reports?***

In order to address the RQ1, we have three projects using a variety of neural network models. The results are given in Table 3, where the outcomes of the model for each project are presented in the table column.

The findings provide us with the following observations: (1) in comparison to not classifying bug reports, and it helps filter out the "uninformative" bug reports when utilizing the LSTM method (when the precision raises). (2) When the LSTM network raises the precision of the classification, it decreases the recall. It

leads to a decrease in the false positives; however, it is also decreasing the true positives. (3) Utilizing the LSTM network, F0.5-measure can be increased while F1-measure could be declined. In terms of the importance of precision and recall, LSTM can not be beneficial. Nonetheless, if in case we choose precision rather than recall under specific trade-off (F0,5-Measure), it is beneficial to use LSTM networks. (4) The performance of the classification accomplish better when utilizing LSTM-network the use of CNN and multilayer perceptron. It demonstrates that the acquired from LSTM long term dependency helps evaluate the finding of the usefulness of the bug report to IR-based bug locating. (5) We also find that, in contrast with Eclipse UI, the output gap between Eclipse SWT and Eclipse JDT is small. It is maybe because we have a smaller training and testing size, which could be one of the possible reasons. Nonetheless, better precision is also provided when F0.5-measure on these two projects is kept higher.

Summary. The LSTM-network shows better results on the three benchmark projects, especially on Eclipse Platform UI. So, we answer *RQ1* that the LSTM-network approach helps filter out "uninformative" bug reports under a particular trade-off (F0.5-Measure) between recall.

7.7.2 ***RQ2: What is the difference between using different neural network models?***

As discussed in Section 3.3, a bug report is classified into a category based on its categorical probabilities, which are computed as output by the Softmax model. That is, based on the output of Softmax, we classify a report as "informative" when $P(\text{“}informative\text{''}) > P(\text{“}uninformative\text{''})$ even the difference is marginal.

To further increase the classification precision, instead of classifying a report to the category with larger probability, we perform classification using a fixed value as the threshold. That is, we classify a bug report as "informative" if $P(\text{“}informative\text{''}) > threshold$ and "uninformative" otherwise. We assume that the bigger threshold, the more confidence of the model in classifying a report as "informative".

We run experiments using different neural network models on the Eclipse Platform UI project by tuning the probability threshold from 0 to 1. For comparison purpose, we introduce a simple baseline approach that classifies a bug report based on the length of its content (summary and description). For the simple baseline approach, we tune the length as the threshold. Then we draw a learning curve for each evaluation metric. The learning curves are shown in Figure 6 for LSTM, Figure 7 for CNN, Figure 8 for multilayer perceptron, and Figure 9 for the simple baseline approach. An overall observation from the results is that the precision increases when we increase the threshold, but the recall drops. When we give more preference to precision, we observe that the F0.5-Measure value also increases.

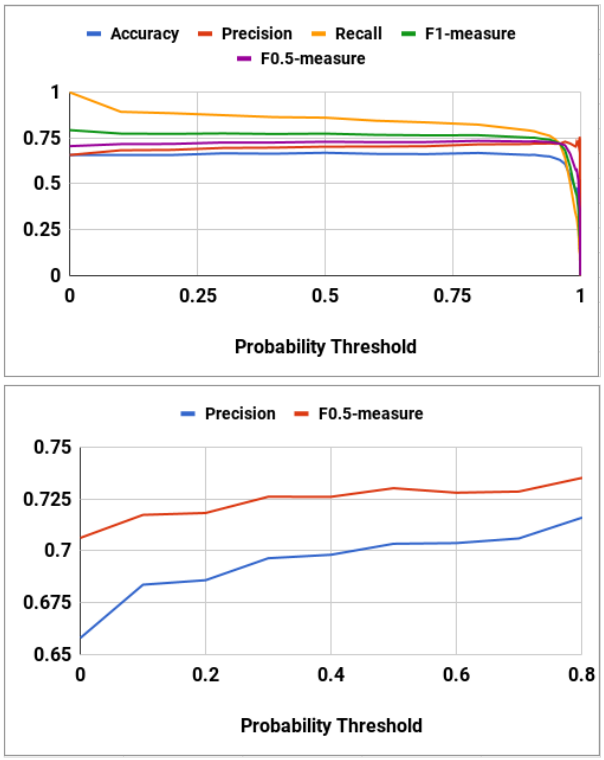

Fig. 6: Learning curves for LSTM.

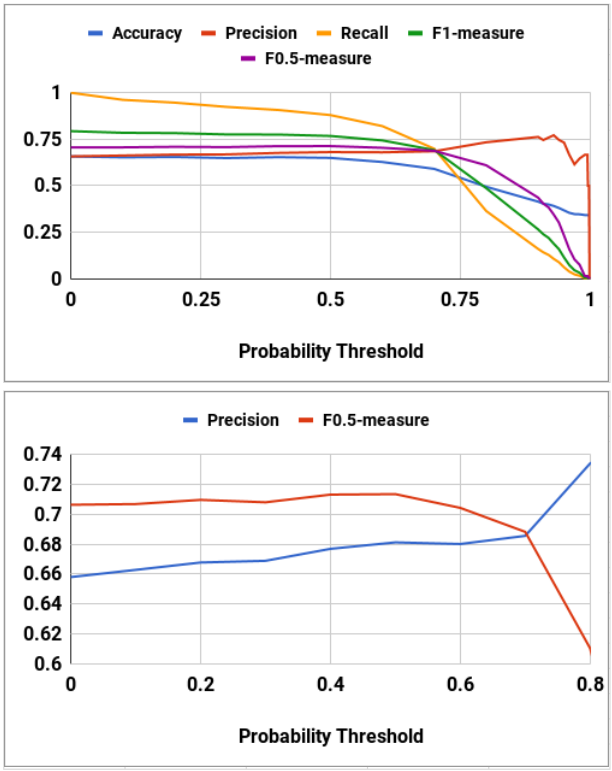

Fig. 7: Learning curves for CNN.

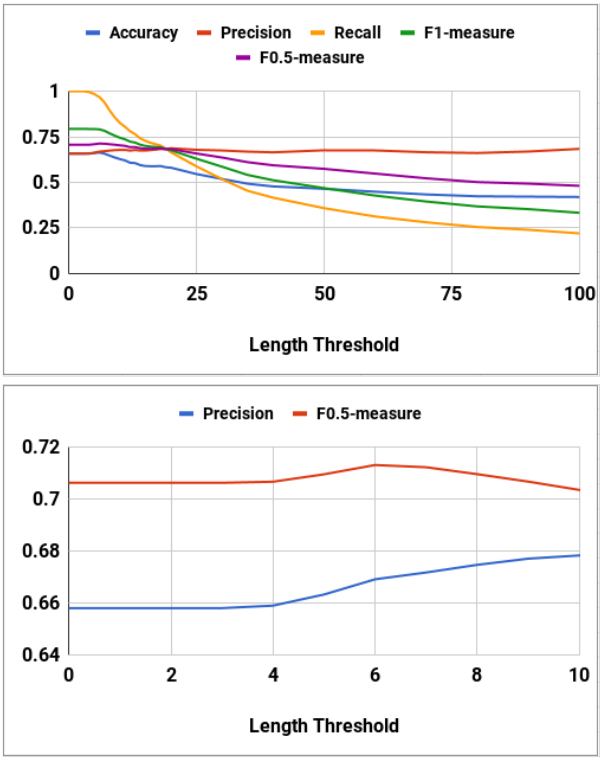

Fig. 9: Learning curves for the simple baseline approach.

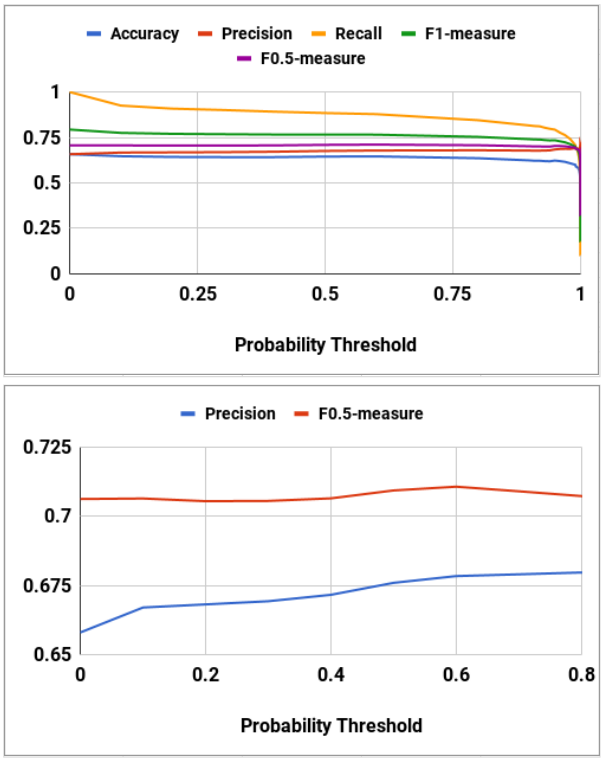

Fig. 8: Learning curves for multilayer perceptron.

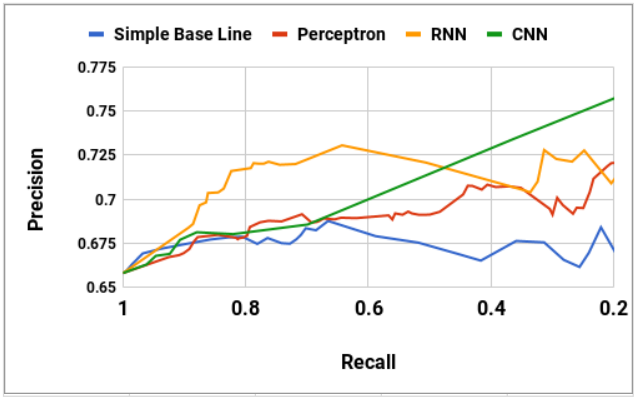

Fig. 10: Precision vs. Recall.

More specifically, take the LSTM result shown in Figure 6. for example, both Precision and F0.5-Measure increase with the probability threshold. When the threshold is set to 0, which means no classifications at all, the Precision is 0.658, and the F0.5-Measure is 0.706. After we increase the threshold to 0.5, we obtain the Precision of 0.703 and F0.5-Measure of 0.73. Then we continue to increase the probability threshold. When the threshold increases to 0.8, which means that we classify a bug re-

port as "informative" only when $P(\text{"}informative\text{''}) > 0.8$, the Precision also increases from 0.703 to 0.716, and the F0.5-Measure increases from 0.73 to 0.735.

We examine Precision changes with the Recall changes in Figure 10 in order to have a better understanding of the variation between models, from which we discover that the LSTM-network (RNN) model achieves higher in Precision performance than other models before the recall declines much further. We do note the increased performance of CNN when the recall is more significant than 0.5 and is low to be helpful.

Summary. We answer *RQ2* that The LSTM network, comparing with CNN and a multilayer perceptron, can accomplish the strongest trade-off between Precision and Recall.

7.7.3 ***RQ3: Can using both the explicit features and the implicit features (neural-network features) help filter out "uninformative" bug reports?***

Table 4 shows the classification results of using different features, where "All Features SVM" means using all the features shown in Table 1 and using SVM as the classifier. These features include explicit features and implicit (neural-network) features. We compared the results of using all features, the results of using only the explicit features, the results of using only the LSTM-network feature, and the results of not doing classification.

As shown in Table 4, using all features achieve the best Accuracy, Precision, F1-Measure, and F0.5-Measure score for all three projects. Take Eclipse Platform UI; for example, compared with not doing classification, using all features for bug report classification helps increase Precision from 0.658 to 0.72, helps increase F1-Measure from 0.794 to 0.81, and helps increase F0.5-Measure from 0.706 to 0.88.

When using all features, we also evaluate the performance of different classifiers. The results are shown in Table 5. SVM and K-Nearest Neighbours (KNN) outperform Decision Tree and Naive Bayes. SVM achieves better Recall and F-Measure scores. KNN achieves better Precision.

Summary. Using both the explicit features and the implicit features helps filter out "uninformative" bug reports. Using all features for bug report classification achieves better Precision and F-Measure score than not doing classification.

7.7.4 ***RQ4: What is the performance of our proposed classification approach in the task of bug localization?***

We use our classification model with all features on bug reports in the research data set to check whether our method helps to enhance the IR-based bug locating system LRWE, and we run the LRWE bug locator system in only "informative" reports. Table 6 shows the comparison results with not doing bug report classification. We observed a noticeable performance improvement in terms of MAP and MRR over not classifying for all three projects.

Summary. Our proposed classification approach, under the trade-off (F1-Measure and F0.5-Measure) between Precision and Recall, helps improve the IR-based bug location system's ranking performance.

8 Threats to Validity

In this section, we identify several threats to the validity of our study.

Construct Threats. One of the main threats is related to our dataset i.e., any mistake in the correspondence between bug reports and their corresponding fixed files would trigger errors in our classification. To mitigate this issue, we did adopt our previous dataset (Ye et al, 2014) because of its reliability and wide adoption by the community (Ye et al, 2014, 2016; Zhou et al, 2012).

Also, any mistake with the labeling implementation could be lethal to our work. To mitigate this, we have randomly tested a few samples of automatically labeled reports, and we reproduced the localization process, and verified that the report is labeled informative if its infected file(s) appear(s) in the top ranked candidates, and uninformative otherwise.

Internal Threat. the internal threats are linked to possible errors in the studies. We have reviewed our developed code, but no faults can be identified.

Another main issue with our classification is the potential over-fitting of our model. Our approach evaluation does not only rely on precision and recall, which are not good indicators of such phnomenon, but also we have tested our model in a more practical aspect by validating the accuracy of localizers when being fed with informative reports. So our model evaluation mainly relies on the performance of these localizers, which is not sensitive to over-fitting.

Table 4: Classification results of using different features and using SVM as the classifier: All Features refers to using both the explicit and implicit (neural-network) features, and NC (No Classification) refers to classifying all the instances (bug reports) as positive ("informative").

Project	Metric	All Features SVM	Explicit Features SVM	LSTM	NC
Eclipse Platform UI	Accuracy	**0.71**	0.665	0.670	0.658
	Precision	**0.72**	0.672	0.703	0.658
	Recall	0.93	0.960	0.862	**1**
	F1-Measure	**0.81**	0.790	0.775	0.794
	F0.5-Measure	**0.88**	0.884	0.730	0.706
SWT	Accuracy	**0.736**	0.714	0.694	0.692
	Precision	0.743	**0.748**	0.694	0.692
	Recall	0.951	0.891	**1**	**1**
	F1-Measure	**0.834**	0.813	0.819	0.818
	F0.5-Measure	**0.9**	0.858	0.739	0.738
JDT	Accuracy	**0.78**	0.769	0.763	0.759
	Precision	**0.777**	0.768	0.762	0.759
	Recall	**1**	0.997	**1**	**1**
	F1-Measure	**0.88**	0.868	0.865	0.863
	F0.5-Measure	**0.944**	0.941	0.8	0.797

Table 5: Classification results of using SVM and difference features: All Features refers to using both the explicit and implicit (neural-network) features, and NC (No Classification) refers to classifying all the instances (bug reports) as positive ("informative").

Project	Metric	All Features SVM	All Features KNN	All Features Decision Tree	All Features Naive Bayes
Eclipse Platform UI	Accuracy	0.71	**0.726**	0.679	0.661
	Precision	0.72	**0.754**	0.708	0.704
	Recall	**0.93**	0.867	0.872	0.836
	F1-Measure	**0.81**	0.806	0.781	0.764
	F0.5-Measure	**0.88**	0.841	0.834	0.806
SWT	Accuracy	0.736	**0.748**	0.719	0.694
	Precision	0.743	**0.765**	0.735	0.732
	Recall	**0.951**	0.923	0.934	0.887
	F1-Measure	0.834	**0.836**	0.823	0.802
	F0.5-Measure	**0.9**	0.886	0.886	0.851
JDT	Accuracy	0.78	**0.786**	0.775	0.742
	Precision	0.777	**0.795**	0.78	0.778
	Recall	**1**	0.967	0.98	0.925
	F1-Measure	**0.88**	0.873	0.869	0.845
	F0.5-Measure	**0.944**	0.927	0.932	0.891

Conclusion. Our conclusions are based on metrics as evaluation scales often used in bug location, we used MAP, and MRR to evaluate bug location efficiency. Besides, these metrics are heavily used in previous studies related to bug localization and assignment (Huo and Li, 2017; Huo et al, 2016; Lam et al, 2015; Saha et al, 2013; Ye et al, 2014, 2016; Zhou et al, 2012).

External Threat. external validity threats apply to our observations being generalized. More than 9,000 bug reports from open-source projects (Ye et al, 2016) Eclipse UI, SWT, and JDT are included in our experiments. Studies in these datasets demonstrate that our model works best when it integrates all knowledge based on interaction and location. We performed across project training and testing, and we were able to develop a model that is project agnostic. This is in favor of the generazibility of our approach. However, it would be interesting to validate in the future, on whether using one single project for designing the classifier would potentially give better results, since projects tend to have their own characteristics that can be relevant for one project and irrelevant for another one.

Further, We aim in the future by expanding the number of projects and the size of bug reports with different programming languages than Java in order to minimize such risks to external validity.

Table 6: Bug locating results: NC (No Classification) refers to classifying all the instances (bug reports) as positive ("informative").

Project	Metric	All Features SVM	All Features KNN	NC
Eclipse Platform UI	MAP	0.434	**0.462**	0.369
	MRR	0.488	**0.515**	0.419
	F1-Measure	**0.81**	0.806	0.794
	F0.5-Measure	**0.88**	0.841	0.706
SWT	MAP	0.456	**0.464**	0.382
	MRR	0.528	**0.537**	0.443
	F1-Measure	0.834	**0.836**	0.818
	F0.5-Measure	**0.9**	0.886	0.738
JDT	MAP	0.443	**0.453**	0.425
	MRR	0.527	**0.541**	0.516
	F1-Measure	**0.88**	0.873	0.863
	F0.5-Measure	**0.944**	0.927	0.797

9 Related Work

In this section, we present related work on different aspects of bug localization. We discuss four areas of related studies, which are IR-based bug localization, other uses of neural networks in software engineering, and briefly touch on non-IR-based bug localization approaches to bug reports and other related studies.

9.1 IR-Based Bug Localization:

Several IR-based bug localizing methods have been suggested using information retrieval techniques to reformulate queries (Chaparro et al, 2019; Rahman and Roy, 2018; Koyuncu et al, 2019), improve the performance of bug localization (Lee et al, 2018; Tantithamthavorn et al, 2018), the relation between bug report and source code (Khatiwada et al, 2018; Le et al, 2017) and clustering bug report and program elements (Hoang et al, 2018).

Lam et al (2015) performed an empirical study to improve bug report handling by automating the task of associating buggy files with a bug report. The authors combined rSVM information retrieval with deep neural networks to associate terms in bug reports to terms in source files. Their model can recommend source code files containing the bug mentioned in a bug report. Huo and Li (2017); Huo et al (2016) recommended multiple methods suggested in a bug report to find faulty source files. The authors proposed a novel convolutional neural network NP-CNN that leverages the structural information of source code in addition to the lexical information to accomplish this task. They followed with another model LS-CNN that combines CNN and LSTM to utilize the sequential information of source code additionally.

Ye et al (2016, 2014) Built an learning-to-rank model to combine multiple features in the ranking of source files for bug reports. iThe model can be trained by utilizing source code files, API code details, bug history, and code change historical information from bug reports that have already been resolved.

Zhou et al (2012) proposed a BugLocator that used an IR-Based technique for ranking the files that show similarity in the textual between source code and bug reports utilizing rSVM. Saha et al (2013) outperforms BugLocator with BLUiR that uses structural information of code to enable more accurate bug localization.

Kim et al (2013) Present an approach that recommends files where a bug is most possibly resolved based on the quality of the bug report. In order to improve localization accuracy, they added an initial phase where bug reports are classified as informative or deficient based on a prediction history of resolved bug reports.

Lukins et al (2010) applied an LDA technique that is accurate and scaleable for automatic bug localization. Nguyen et al (2011) proposed an automated approach that assists developers to reduce the effects. They performed their approach by narrowing the search space of bug files. They developed LDA techniques to represent technical aspects in the content of both documents bug reports and source files.

Rao and Kak (2011) compared five IR-Based models for bug localization. The models are VSM, LSA, UM, CBDM, and LDA. Their results show that the more complex models (LDA, LSA, CBDM) have not outperformed simpler text models suhc as (UM, VSM) for bug localization.

9.2 Using Neural Networks to Support Software Engineering

Effort estimation is necessary for planning and managing a software project. Choetkiertikul et al (2018) utilized deep learning with long short-term memory and recurrent highway network to facilitate effort estimation for agile projects. They used deep learning to model and predict estimations of story points, a unit of measure for the effort to complete a user story, or resolve an issue.

Developers often need to utilize APIs to implement functionality, but it can be a significant obstacle to deal with unfamiliar libraries or frameworks. Gu et al (2016) utilized RNN Encoder-Decoder for a deep learning approach called DeepAPI. DeepAPI allows a natural language query to accurately generate a relevant API sequence.

Online developer forums are full of individual units of programmer knowledge that have the potential to be linked for being related, duplicates, etc. Xu et al (2016) utilized word embeddings and convolutional neural networks for a deep learning. Their study is based on the approach to semantically linking knowledge units on StackOverflow that outperformed traditional methods. Fu and Menzies (2017) followed up this approach with a differential evolution approach that achieves similar results on the scale of minutes rather than hours with the deep learning approach. They showed that deep learning may provide benefits for software engineering, but simpler or faster methods should still be considered.

9.3 Non-IR-Based Bug Localization

Information retrieval approaches are not the only way to try handling bug reports. Some approaches are not IR-based or augment/combine with IR to accomplish bug report handling tasks. Cleve and Zeller (2005) focused on cause transitions to find locations of defects. Dit et al (2013) utilized web mining algorithms to analyze execution information. Poshyvanyk et al (2013, 2007) have utilized both Formal Concept Analysis and scenario-based probabilistic ranking of events. Liu et al (2005) used a model based on pattern evaluation between correct and incorrect runs to quantify bug-relevance. Jin and Orso (2013) used synthesized passing and failing executions to perform fault localization. Le et al (2015); B. Le et al (2016) utilized approaches to program spectra analysis to find suspicious words and invariant mining. Jones and Harrold (2005) implemented Tarantula approach of generating likelihood/suspicion for each statement of source code using the code entities executed bypassing and failing test cases.

9.4 Other Related Studies.

Bug Severity. Determining the severity of bug reports automatically is another area where handling of bug reports can be improved. Lamkanfi et al (2010) used a Naïve Bayes based approach to investigate if severity can be accurately predicted. They concluded that a sufficient training set can achieve reasonable prediction accuracy. Zhang et al (2016) described a system to find similar historical bug reports utilizing a modified REP algorithm and K-Nearest Neighbor. Then, an improved performance severity prediction algorithm was developed with the extracted features of the bug reports.

Bug Triage. Another direction for reducing the effort of handling bug reports is to automate triage of bug reports to the developer(s) that are likely to resolve them. Anvik et al (2006); Anvik and Murphy (2011) used support vector machines and other machine learning approaches to implement developer recommending models achieving varying degrees of precision. Hu et al (2014) implemented a recommendation method called Bug Fixer that utilizes historical information of source code components where developers have fixed bugs previously. Zhang and Lee (2013) implemented a hybrid system that utilizes the unigram model to find similar bug reports and then recommends a developer based on the developer's probability to fix and a model of the developer's activity and experience. Bhattacharya et al (2012) studied a variety of machine learning tools and graphs that used to assign bugs reliably to developers. Xuan et al (2015) utilized an instance and feature selection model defined by historical bug datasets to decrease system size and increase bug triage accuracy. Shokripour et al (2013) used an approach that used noun extraction and simple term weighting to predict bug location and then used a location-based approach to recommend assignment of the bug to a developer.

10 Conclusions and Future Work

This paper presents an approach that includes both explicit and implicit features learned from neural network models to classify a bug report as "informative" or "uninformative". Although a "informative" report is considered to be useful for detecting the bug for an IR-based bug location system, it is considered that an "uninformative" report for an IR system does not support and is discarded. The results of the evaluation indicate that our proposed classification model leads to filtering "uninformative" reports and enhances the ranking efficiency of a state of the art IR system in software bugs localization. The LSTM network delivers

better f-measurements than CNN and multi-layer perceptrons among different neural network models. With both explicit and implicit features, precision and F-measurement produce the best results compared to explicit features, with only the LSTM feature and not classified. We concluded that we could make a better tradeoff between precision and Recall by filtering out "uninformative" reports. In addition, our classification model improves the performance of bug localization tasks in that MAP and MRR, particularly in cases where precision is more critical than Recall.

For future research, further IR-based bug location systems on more software projects would be used to test the efficacy of our approach. Meanwhile, we plan to investigate alternative methods for translating bug reports into vector representations such as *sent2vec* (Pagliardini et al, 2017). Also, we intend to review numerous bug reports manually in order to have an overview on what kind of quality that makes an IR system useful. We aim to build features that reflect the quality of bug reports effectively and merge them with both the neural network-based features for accurate classification.

Compliance with ethical standards

Disclosure of funding. This study was partially funded by Rochester Institute of Technology, SRS Proposal Number 17080438. This work was carried out at both California State University and Rochester Institute of Technology.

Conflict of interest. The authors declare that they have no conflict of interest.

Ethical approval. This article does not contain any studies with human participants or animals performed by any of the authors.

References

Anvik J, Murphy GC (2011) Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Trans Softw Eng Methodol 20:10:1–10:35

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, New York, NY, USA, ICSE '06, pp 361–370

B Le TD, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp 177–188, DOI 10.1145/2931037.2931049, URL http://doi.acm.org/10.1145/2931037.2931049

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, ICSE '13, pp 712–721, URL http://dl.acm.org/citation.cfm?id=2486788.2486882

Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. Trans Neur Netw 5(2):157–166, DOI 10.1109/72.279181, URL http://dx.doi.org/10.1109/72.279181

Bettenburg N, Just S, Schröter A, Weiss C, Premraj R, Zimmermann T (2008) What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, SIGSOFT '08/FSE-16, pp 308–318

Bhattacharya P, Neamtiu I, Shelton CR (2012) Automated, highly-accurate, bug assignment using machine learning and tossing graphs. J Syst Softw 85(10):2275–2292, DOI 10.1016/j.jss.2012.04.053, URL http://dx.doi.org/10.1016/j.jss.2012.04.053

Breu S, Premraj R, Sillito J, Zimmermann T (2010) Information needs in bug reports: Improving cooperation between developers and users. In: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, New York, NY, USA, CSCW '10, pp 301–310

Bruegge B, Dutoit AH (2009) Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd edn. Prentice Hall Press, Upper Saddle River, NJ, USA

Buse RPL, Zimmermann T (2012) Information needs for software development analytics. In: Proceedings of the 2012 International Conference on Software Engineering, Piscataway, NJ, USA, ICSE 2012, pp 987–996

Chaparro O, Florez JM, Marcus A (2019) Using bug descriptions to reformulate queries during text-retrieval-based bug localization. Empirical Software Engineering 24(5):2947–3007

Choetkiertikul M, Dam HK, Tran T, Pham TTM, Ghose A, Menzies T (2018) A deep learning model for estimating story points. IEEE Transactions on Software Engineering PP(99):1–1, DOI 10.1109/TSE.2018.2792473

Cleve H, Zeller A (2005) Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, ICSE '05, pp 342–351

Dilshener T, Wermelinger M, Yu Y (2016) Locating bugs without looking back. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp 286–290, DOI 10.1109/MSR.2016.037

Dit B, Revelle M, Poshyvanyk D (2013) Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empirical Softw Engg 18(2):277–309

Fu W, Menzies T (2017) Easy over hard: A case study on deep learning. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2017, pp 49–60, DOI 10.1145/3106237.3106256, URL http://doi.acm.org/10.1145/3106237.3106256

Gal Y, Ghahramani Z (2016) A theoretically grounded application of dropout in recurrent neural networks. In: Proceedings of the 30th International Conference on Neural Information Processing Systems, Curran Associates Inc., USA, NIPS'16, pp 1027–1035, URL http://dl.acm.org/citation.cfm?id=3157096.3157211

Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010, pp 249–256, URL http://www.jmlr.org/proceedings/papers/v9/glorot10a.html

Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. The MIT Press

Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2016, pp 631–642, DOI 10.1145/2950290.2950334, URL http://doi.acm.org/10.1145/2950290.2950334

Hoang T, Oentaryo RJ, Le TDB, Lo D (2018) Network-clustered multi-modal bug localization. IEEE Transactions on Software Engineering 45(10):1002–1023

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780, DOI 10.1162/neco.1997.9.8.1735, URL http://dx.doi.org/10.1162/neco.1997.9.8.1735

Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ASE '07, pp 34–43

Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. Neural Netw 2(5):359–366, DOI 10.1016/0893-6080(89)90020-8, URL http://dx.doi.org/10.1016/0893-6080(89)90020-8

Hu H, Zhang H, Xuan J, Sun W (2014) Effective bug triage based on historical bug-fix information. 2014 IEEE 25th International Symposium on Software Reliability Engineering pp 122–132

Huo X, Li M (2017) Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, AAAI Press, IJCAI'17, pp 1909–1915, URL http://dl.acm.org/citation.cfm?id=3172077.3172153

Huo X, Li M, Zhou ZH (2016) Learning unified features from natural and programming languages for locating buggy source code. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, AAAI Press, IJCAI'16, pp 1606–1612, URL http://dl.acm.org/citation.cfm?id=3060832.3060845

Jin W, Orso A (2013) F3: Fault localization for field failures. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, New York, NY, USA, ISSTA 2013, pp 213–223

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ASE '05, pp 273–282

Khatiwada S, Tushev M, Mahmoud A (2018) Just enough semantics: An information theoretic approach for ir-based software bug localization. Information and Software Technology 93:45–57

Kim D, Tao Y, Kim S, Zeller A (2013) Where should we fix this bug? A two-phase recommendation model. IEEE Trans Softw Eng 39(11):1597–1610

Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. CoRR abs/1412.6980, URL http://arxiv.org/abs/1412.6980, 1412.6980

Koyuncu A, Bissyandé TF, Kim D, Liu K, Klein J, Monperrus M, Traon YL (2019) D&c: A divide-and-conquer approach to ir-based bug localization. arXiv preprint arXiv:190202703

Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2015) Combining deep learning with information retrieval to localize buggy files for bug reports (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 476–481, DOI 10.1109/ASE.2015.73

Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2017) Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp 218–229, DOI 10.1109/ICPC.2017.24

Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp 1–10, DOI 10.1109/MSR.2010.5463284

LaToza TD, Myers BA (2010) Hard-to-answer questions about code. In: Evaluation and Usability of Programming Languages and Tools, New York, NY, USA, PLATEAU '10, pp 8:1–8:6

Le QV, Mikolov T (2014) Distributed representations of sentences and documents. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, pp 1188–1196, URL http://jmlr.org/proceedings/papers/v32/le14.html

Le TDB, Oentaryo RJ, Lo D (2015) Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 579–590, DOI 10.1145/2786805.2786880, URL http://doi.acm.org/10.1145/2786805.2786880

Le TDB, Thung F, Lo D (2017) Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools. Empirical Software Engineering 22(4):2237–2279

Lecun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11):2278–2324, DOI 10.1109/5.726791

Lee J, Kim D, Bissyandé TF, Jung W, Le Traon Y (2018) Bench4bl: reproducibility study on the performance of ir-based bug localization. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 61–72

Levy O, Goldberg Y (2014) Neural word embedding as implicit matrix factorization. In: Ghahramani Z, Welling M, Cortes C, Lawrence N, Weinberger K (eds) Advances in Neural Information Processing Systems 27, Curran Associates, Inc., pp 2177–2185, URL http://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf

Liu C, Yan X, Fei L, Han J, Midkiff SP (2005) Sober: Statistical model-based bug localization. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, ESEC/FSE-13, pp 286–295

Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using Latent Dirichlet Allocation. Inf Softw Technol 52(9):972–990

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA

Miller GA (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. The Psychological Review 63(2):81–97

Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings Eighth Working Conference on Reverse Engineering, pp 13–22, DOI 10.1109/WCRE.2001.957806

Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI Press, AAAI'16, pp 1287–1293, URL http://dl.acm.org/citation.cfm?id=3015812.3016002

Murphy-Hill E, Zimmermann T, Bird C, Nagappan N (2013) The design of bug fixes. In: Proceedings of the 2013 International Conference on Software Engineering, Piscataway, NJ, USA, ICSE '13, pp 332–341, URL http://dl.acm.org/citation.cfm?id=2486788.2486833

Nair V, Hinton GE (2010) Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on International Conference on Machine Learning, Omnipress, USA, ICML'10, pp 807–814, URL http://dl.acm.org/citation.cfm?id=3104322.3104425

Nguyen AT, Nguyen TT, Al-Kofahi J, Nguyen HV, Nguyen TN (2011) A topic-based approach for narrowing the search space of buggy files from a bug report. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, ASE '11, pp 263–272, DOI 10.1109/ASE.2011.6100062

Pagliardini M, Gupta P, Jaggi M (2017) Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features. ArXiv e-prints 1703.02507

Poshyvanyk D, Gueheneuc YG, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng 33(6):420–432

Poshyvanyk D, Gethers M, Marcus A (2013) Concept location using formal concept analysis and information retrieval. ACM Trans Softw Eng Methodol 21(4):23:1–23:34

Rahman MM, Roy CK (2018) Improving ir-based bug localization with context-aware query reformulation. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and

Symposium on the Foundations of Software Engineering, pp 621–632

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, New York, NY, USA, MSR '11, pp 43–52

Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L (2014) ImageNet Large Scale Visual Recognition Challenge. ArXiv e-prints 1409.0575

Saha R, Lease M, Khurshid S, Perry D (2013) Improving bug localization using structured information retrieval. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp 345–355

Shokripour R, Anvik J, Kasirun ZM, Zamani S (2013) Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In: Proceedings of the 10th Working Conference on Mining Software Repositories, Piscataway, NJ, USA, MSR '13, pp 2–11, URL http://dl.acm.org/citation.cfm?id=2487085.2487089

Tantithamthavorn C, Abebe SL, Hassan AE, Ihara A, Matsumoto K (2018) The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. Information and Software Technology 102:160–174

Voorhees EM (1999) The TREC-8 question answering track report. In: In Proceedings of TREC-8, pp 77–82

Xu B, Ye D, Xing Z, Xia X, Chen G, Li S (2016) Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2016, pp 51–62, DOI 10.1145/2970276.2970357, URL http://doi.acm.org/10.1145/2970276.2970357

Xuan J, Jiang H, Hu Y, Ren Z, Zou W, Luo Z, Wu X (2015) Towards effective bug triage with software data reduction techniques. IEEE Transactions on Knowledge and Data Engineering 27:264–280

Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, FSE 2014, pp 689–699, URL http://dl.acm.org/citation.cfm?id=2337223.2337226

Ye X, Shen H, Ma X, Bunescu R, Liu C (2016) From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '16, pp 404–415, DOI 10.1145/2884781.2884862, URL http://doi.acm.org/10.1145/2884781.2884862

Zhang T, Lee B (2013) A hybrid bug triage algorithm for developer recommendation. In: SAC

Zhang T, Chen J, Yang G, Lee B, Luo X (2016) Towards more accurate severity prediction and fixer recommendation of software bugs. J Syst Softw 117(C):166–184, DOI 10.1016/j.jss.2016.02.034, URL https://doi.org/10.1016/j.jss.2016.02.034

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 2012 International Conference on Software Engineering, Piscataway, NJ, USA, ICSE 2012, pp 14–24, URL http://dl.acm.org/citation.cfm?id=2337223.2337226