Information and Software Technology 105 (2019) 17–29

Contents lists available at ScienceDirect

Information and Software Technology

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

Improving bug localization with word embedding and enhanced convolutional neural networks

Yan Xiao[*], Jacky Keung, Kwabena E. Bennin, Qing Mi

Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China

ARTICLE INFO

Keywords:
Bug localization
Convolutional neural network
Word embedding
TF-IDF
Deep learning
Semantic information

ABSTRACT

Context: Automatic localization of buggy files can speed up the process of bug fixing to improve the efficiency and productivity of software quality assurance teams. Useful semantic information is available in bug reports and source code, but it is usually underutilized by existing bug localization approaches.

Objective: To improve the performance of bug localization, we propose DeepLoc, a novel deep learning-based model that makes full use of semantic information.

Method: DeepLoc is composed of an enhanced convolutional neural network (CNN) that considers bug-fixing recency and frequency, together with word-embedding and feature-detecting techniques. DeepLoc uses word embeddings to represent the words in bug reports and source files that retain their semantic information, and different CNNs to detect features from them. DeepLoc is evaluated on over 18,500 bug reports extracted from AspectJ, Eclipse, JDT, SWT, and Tomcat projects.

Results: The experimental results show that DeepLoc achieves 10.87%–13.4% higher MAP (mean average precision) than conventional CNN. DeepLoc outperforms four current state-of-the-art approaches (DeepLocator, HyLoc, LR + WE, and BugLocator) in terms of Accuracy@k (the percentage of bug reports for which at least one real buggy file is located within the top *k* rank), MAP, and MRR (mean reciprocal rank) using less computation time.

Conclusion: DeepLoc is capable of automatically connecting bug reports to the corresponding buggy files and achieves better performance than four state-of-the-art approaches based on a deep understanding of semantics in bug reports and source code.

1. Introduction

Bug localization is a significant task during software maintenance. To locate a newly reported bug, developers must carefully analyze the bug report and review numerous source code files. Thus, bug resolution activity usually requires considerable time and effort [23] to improve software quality and ensure software integrity [5]. To reduce maintenance costs and substantially improve the efficiency and productivity of the whole software team, researchers have proposed several automated bug localization approaches and tools for localization of potential buggy files, especially when the project is large and complex and involves thousands of source files.

The existing bug localization techniques can be categorized into four main groups. The first group comprises approaches that use traditional features related to program analysis information [14], such as passing or failing execution information with test cases. Spectrum-based fault localization techniques [1,25] are typical examples of these approaches. These methods extract static features from the source code or execution information, which is a time-consuming process. The second group comprises the information retrieval (IR)-based approaches, which search and rank buggy files for a given bug report. These approaches measure the textual similarity between bug reports and the names of classes or methods in source code files. This kind of similarity focuses mainly on the term weights as used in the IR field [48]. In addition, machine learning (ML)-based approaches have been proposed. These approaches adopt ML models that are trained to match the topics of bug reports with those of source files [34] or classify source files into multiple classes using formerly fixed files [16].

The approaches in the second and third groups focus on the term weights of natural language texts and do not consider source code semantics during the bug localization process. According to prior research, semantic information is important for code suggestion [33] and

[*] Corresponding author.
E-mail addresses: yanxiao6-c@my.cityu.edu.hk (Y. Xiao), Jacky.Keung@cityu.edu.hk (J. Keung), kebennin2-c@my.cityu.edu.hk (K.E. Bennin), Qing.Mi@my.cityu.edu.hk (Q. Mi).

https://doi.org/10.1016/j.infsof.2018.08.002
Received 27 December 2017; Received in revised form 11 July 2018; Accepted 7 August 2018
Available online 15 August 2018

code completion [11] and is also useful for bug localization [49]. To bridge the semantic gap, deep learning has been introduced into the domain of bug localization for semantic parsing. Deep learning is known to have excellent performance in natural language processing (NLP) and image processing [7,21]. The recently proposed deep learning-based model for bug localization combines deep neural networks (DNNs) and IR techniques [22]. Its results are influenced to some extent by the performance of IR techniques. Most importantly, many DNN models are used together, which makes it very complex and difficult to accurately adjust the parameters of the model.

To extract the underlying semantic information from bug reports and source code, this paper proposes a CNN-based model for bug localization, namely DeepLoc. CNN can extract local features by convolving filters and performs well in semantic parsing-related NLP problems [17]. Thus, DeepLoc adopts CNN to extract features from vectors of bug reports and source files converted using word-embedding techniques. Moreover, DeepLoc learns to correlate bug reports to the corresponding buggy files using an enhanced CNN, rather than textual similarity, which is used in IR-based approaches. We conduct empirical experiments on five project datasets to show the effect of the enhanced CNN on DeepLoc and evaluate DeepLoc's performance by comparing it to four state-of-the-art bug localization approaches: DeepLocator, HyLoc, LR + WE, and BugLocator. All these competitors were proposed to localize buggy files for bug reports, which were based on the textual similarity rather than semantic information of the words and phrases.

The main contributions of this paper are the following:

- We transform all code tokens from source files into vectors using word-embedding techniques for bug localization. The visualization of the learned vectors indicates that this method can benefit the proposed model.
- We develop DeepLoc, which consists of word embedding, feature detection, and a proposed enhanced CNN. The enhanced CNN uses important bug-fixing experience (bug-fixing recency and frequency) and semantic information that can be extracted by a conventional CNN from bug reports and source files.
- A set of experiments are conducted to validate DeepLoc's feasibility and effectiveness. We make our dataset publicly available and provide the tool used to construct the dataset[1].

In this work, we enhance our previous model, DeepLocator, presented in [47]. DeepLocator first pre-treats bug reports and source files using revised TF-IDuF and Abstract Syntax Tree (AST) detection, respectively. The same word-embedding technique (word2vec) is then applied to transform the preprocessed words from bug reports and source files into vectors. Later, the features of these vectors are extracted by one CNN. However, the use of the same word-embedding technique and CNN to deal with bug reports and source files ignored the differences between them and thus limited the performance of DeepLocator.

Although the motivation for this work is somewhat derived from Xiao et al. [47], the fundamental differences between the two studies are as follows. (1) This paper uses two different word-embedding techniques (Sent2Vec and combined word2vec) to convert bug reports and source files, respectively, into vectors. (2) All code tokens are transformed in this paper, instead of some AST nodes parsed from the source files in [47]. (3) Two CNNs are used to extract features from the vectors of bug reports and source files before they are supplied to the proposed enhanced CNN. (4) We report better results and new analyses and show the visualization of the learned vectors to demonstrate the effectiveness of the word-embedding techniques for bug localization.

The remainder of this paper is structured as follows. Section 2 reviews the preliminary knowledge related to DeepLoc. Section 3 gives a detailed description of the proposed model. The experimental results are presented in Section 4 and the discussion is presented in Section 5. The threats to validity are discussed in Section 6, and related works are reviewed in Section 7. We conclude the paper and discuss directions for future work in Section 8.

[1] https://github.com/yanxiao6/BugLocalization-dataset.

2. Preliminaries

2.1. Term frequency-Inverse document frequency (TF-IDF)

The Term Frequency-Inverse Document Frequency (TF-IDF) technique is widely used in text mining and information retrieval (IR). TF represents the frequency with which a term appears in a document, and IDF denotes the inverse of the number of documents in which the term appears in the entire corpus. TF-IDF can filter some common terms in documents in specified repositories. The formulas for TF-IDF are as follows [48]:

$$\begin{aligned} tf_{t,d} &= f_{t,d} \quad idf_{t,d} = log\left(\frac{N}{df_t}\right) \\ w_{tf-idf} &= tf_{t,d} \times idf_{t,d} \end{aligned} \tag{1}$$

where $f_{t,\ d}$ is the frequency of term t that appears in document d, N refers to the total number of documents in the document collections, and df_t reflects the number of documents in the corpus containing term t. However, this kind of general corpus is not always available in practice or for research, and the equations represent general behaviors that ignore some important individual information.

To address the aforementioned problems, Beel et al. [2] presented term frequency-user focused inverse document frequency (TF-IDuF) using personal document repositories instead of a general corpus. TF-IDuF is defined as follows [2]:

$$\begin{aligned} tf_{t,d} &= f_{t,d} \quad idf_{t,d} = log\left(\frac{N_u}{n_{t,u}}\right) \\ w_{tf-idf} &= tf_{t,d} \times idf_{t,d} \end{aligned} \tag{2}$$

where N_u refers to the total number of documents in the user collections and $n_{t,\ u}$ reflects the number of documents in the collections that incorporate term t.

Eqs. (1) and (2) are fundamental for term weights. When large differences in frequencies are present, the term weights computed by (1) and (2) do not achieve good performance in practice [8]. Therefore, it is necessary to dampen the effects of large differences in frequencies, which is further discussed below.

The logarithm variant of TF can improve the performance of TF-IDF in practice, as demonstrated in Croft's experiments [8]. Specifically, it can dampen the effects of large differences in frequencies:

$$tf_{t,d} = log(f_{t,d}) + 1 \tag{3}$$

Inspired by Croft et al.'s work [8], we propose an rTF-IDuF to further improve the model's performance:

$$w_{tf-idf} = (log(f_{t,d}) + 1) \times \log\left(\frac{N_u}{n_{t,u}}\right) \tag{4}$$

where $f_{t,\ d}$ is the frequency of term t appearing in document d, N_u refers to the total number of documents in user collections, and $n_{t,\ u}$ reflects the number of documents in user collections that contain term t.

In reality, a complete corpus is not always available. In addition, the bug reports for different projects usually differ in their writing style and contexts. Therefore, it is convenient and reasonable to apply rTF-IDuF, which focuses on personal collections and is used in this paper.

2.2. Word embedding

Word embedding refers to the distributed representations of words

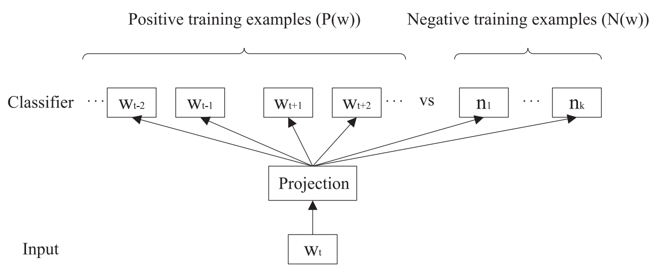

Positive training examples (P(w))
Negative training examples (N(w))
Classifier
Wt-2
Wt-1
Wt+1
Wt+2
vs
n1
nk
Projection
Input
Wt

Fig. 1. The architecture of the Skip-gram model.

in a vector space in which similar words are close to each other [30]. After performing word embedding, words are represented by vectors of real values instead of text, which enables text analysis using various statistical models. Mikolov et al. [29] proposed word2vec techniques for efficient learning of high-quality distributed vector representations. Furthermore, this model can capture precise syntactic and semantic word similarities.

Fig. 1 illustrates the architecture of the Skip-gram model, which uses one of the word2vec techniques. The objective of the Skip-gram model is to learn word representations that maximize the classification of the surrounding words based on a current word in the same sentence. Formally, given a sequence of T training words $w_1, \ldots, w_t, \ldots, w_T$, each current word w_t is used as the input to predict its context words w_{t+r} within a certain range. The Skip-gram model is trained to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le r\le c} \log p(w_{t+r}|w_t) \tag{5}$$

where c is the size of the training context centered at the current word w_t and $r \neq 0$.

Because the cost of computing the derivative of each log probability in Eq. (5) is proportional to the number of words in large vocabulary, Mikolov et al. [30] introduced a negative sampling technique to accelerate the Skip-gram model. Using this technique, its optimization objective is defined as:

$$\frac{1}{T}\sum_{t=1}^{T}\left(\sum_{w_i\in P(w)} \log \sigma(w_i^{\top} w_t) + \sum_{w_j\in N(w)} \log \sigma(-w_j^{\top} w_t)\right) \tag{6}$$

where $\sigma(x) = 1/(1+\exp(-x))$, $P(w)$ is the positive training examples and $N(w)$ is the randomly sampled negative training examples of w_t.

Another word2vec technique proposed by Mikolov et al. [29] is the continuous bag-of-words (CBOW) model, which is similar to the Skip-gram model. However, instead of predicting context words based on the current word, CBOW tries to maximize the classification of the current word based on the context words in the same sentence. Pagliardini et al. proposed Sent2Vec [35], which is an extension of CBOW [29], to compose sentence embedding. Unlike in CBOW, the context words running over the corpus in Sent2Vec are entire sentences instead of fixed-length context words, whose architecture is shown in Fig. 2. w_t in formula (6) is now changed into the entire sentence where w_t is, but w_t is excluded.

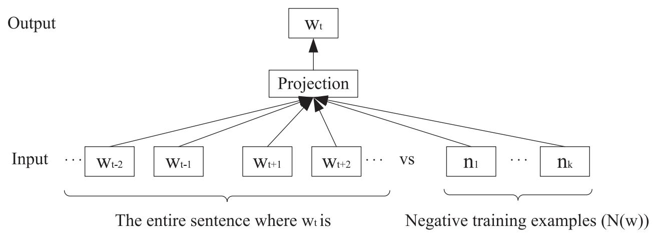

Output
Wt
Projection
Input
Wt-2
Wt-1
Wt+1
Wt+2
vs
n1
nk
The entire sentence where Wt is
Negative training examples (N(w))

Fig. 2. The architecture of Sen2Vec.

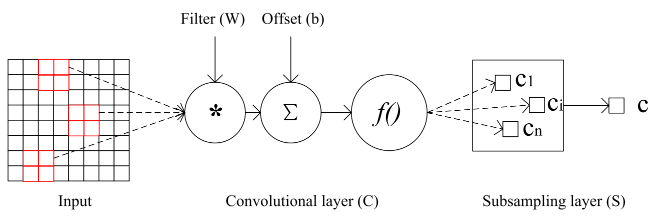


Fig. 3. The process of convolution and subsampling of a CNN.

2.3. Convolutional neural network (CNN)

CNN, inspired by biological processes and multilayer perceptron, is the first learning algorithm based on neocognitron [9] and receptive fields [12]. CNN contains several convolutional layers that are often combined with pooling steps and then followed by a fully-connected layer similar to a standard multilayer neural network [20].

The process of convolution and pooling (subsampling) [41] is shown in Fig. 3. In the process of convolution C, the input is convoluted with a trainable filter W and an added offset b. Later, a nonlinear function is used to obtain a reduced feature map c_i, followed by is the subsampling procedure S. A mean-over-time polling or max-over-time pooling operation [7] is frequently used in this layer, which takes the mean ($c = mean(c_i)$) or maximum value ($c = max(c_i)$) as the feature corresponding to this particular filter. The convolutional layer is used to detect features that are then subsampled in the pooling layer to reduce the number of parameters. The process also alleviates the over-fitting problem [41].

The basic structure of a CNN [21] is shown in Fig. 4. C layers are used to capture features. The input of every neuron in C layers is linked with the local receptive field of the former layer, and the local feature is extracted using convolutional operations. Once the local feature is extracted, its positional relationship with other features is known. S layers are feature mapping layers. Each computational layer has several feature maps, and every feature map is a two-dimension surface. The weight of each neuron in this surface is the same, which follows the well-known parameter sharing scheme. The Rectified Linear Units (ReLU) [20] function is used to increase the network's nonlinear properties.

2.4. Batch normalization

Batch normalization was proposed by Ioffe and Szegedy [13] to reduce internal covariate shift for deep networks. It allows the network to be trained at a much faster learning rate and reduces its sensitivity to initialization. Thus, the use of batch normalization can accelerate the training process more than tenfold while retaining the same or much greater accuracy [13]. Given a mini-batch with m values $(x_1, \ldots, x_i, \ldots, x_m)$, the transforming procedure is as follows:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{7}$$

$$BN(x_i) = \gamma \hat{x}_i + \beta \tag{8}$$

where μ and σ are the mean ($\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$) and variance ($\sigma = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$) of the mini-batch, ϵ is a constant (e.g., 1e-3) added to the variance for the numerical purpose, and γ and β are additional parameters to be learned during training.

3. DeepLoc

In this section, we describe DeepLoc, a deep learning-based model that automatically localizes buggy files for bug reports. DeepLoc uses two conventional CNNs to separately detect features from the vectors of bug reports and source code files converted using two different word-embedding techniques and an enhanced CNN for bug localization.

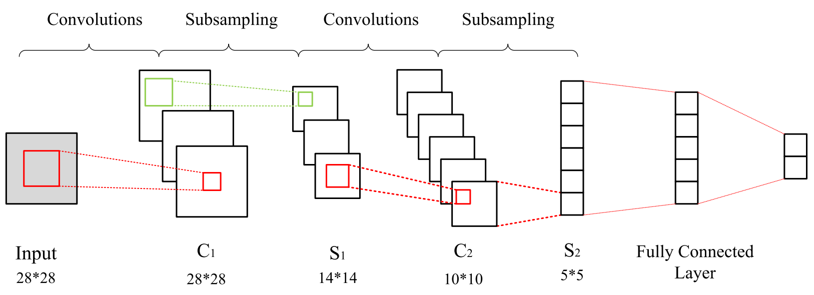


Fig. 4. The net structure of a CNN.

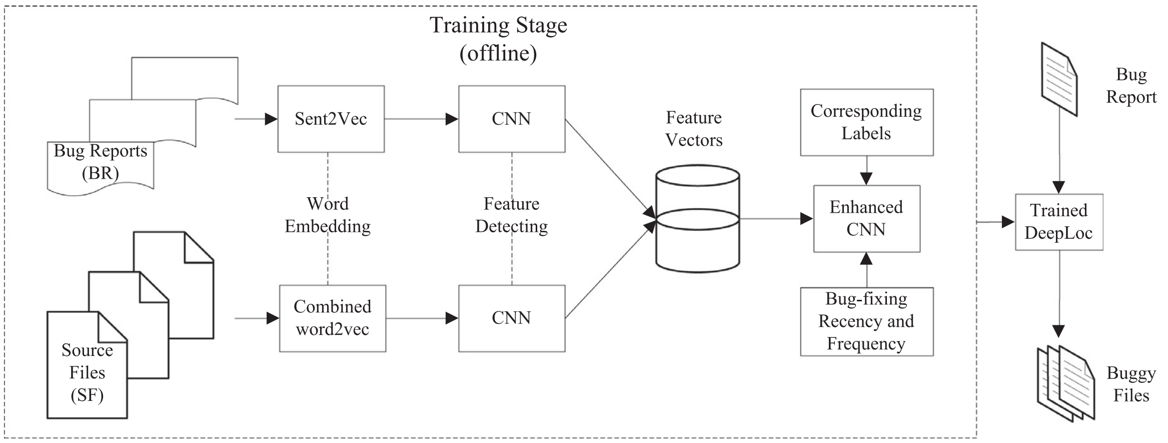


Fig. 5. The overall workflow of DeepLoc.

Fig. 5 provides the overall workflow of DeepLoc. In the training phase, bug reports and source files are first transformed into vectors using two different word-embedding techniques (Sent2Vec [35] and weighted average word2vec [15]). The corresponding vectors are then fed into two conventional CNNs to detect their features. Finally, the features of the bug reports and source files are paralleled as the input to the enhanced CNN, along with corresponding labels (buggy or not buggy) and the bug-fixing histories. The training stage is conducted offline. When we receive new bug reports, the trained model will efficiently utilize the bug reports to obtain the locations of buggy files. The details of DeepLoc are discussed in the following sections.

3.1. Data pre-processing

Developers often combine words to create a new "word" when defining classes or methods in source code files, which can also be mentioned in bug reports. Thus, we preprocess combined words in both bug reports and source files. According to the CamelCase Naming Convention [3], combined words can be split into separate real words based on capital letters. For example, "WorkbenchActionBuilder" is split into "Workbench", "Action" and "Builder". Note that some special capital words, such as URI (Uniform Resource Identifier [28]), IDE (Integrated Development Environment [39]), etc., should not be separated. Finally, we change all capital letters to lowercase letters, to adhere to the rules of word embedding.

3.2. Word embedding

Words themselves cannot be directly input into a CNN [52]. Thus, the pretreated words must be embedded into vectors. Two variants of word-embedding techniques are adopted in this paper to transform words in bug reports and source files into vectors that retain the words' semantics.

As shown in Table 1, bug reports usually contain summaries and descriptions that comprise many words. If all words are transformed into vectors as input for CNNs, the model will be difficult to train due to the redundant input. In addition, a large amount of memory is needed to store these vectors. Fortunately, bug reports are written in natural languages and consist of sentences, so we can convert each sentence in bug reports into a vector instead of transforming each word into a vector. Summaries are always written as one sentence containing condensed and abbreviated information. It is difficult to retain important information if summaries are regarded as sentences. However, descriptions usually contain many sentences and some redundant information. So, to distinguish summaries and descriptions, we convert summaries into vectors based on words using a word2vec technique with the Skip-gram model [29] and transform descriptions into vectors based on sentences using Sent2Vec [35]. Fig. 6 illustrates an example of transforming descriptions in bug reports into vectors using Sent2Vec. Each sentence in a description is transformed into a k-dimensional vector. The dimension of word vectors of each description in a bug report is $n_S \times k$, where n_S is the number of sentences in each description

Table 1
Subject projects [48].

Project	Time range	# of Bug reports	# of Fixed buggy files	Avg. # of Buggy files per bug	Max. # of Words per bug report	Avg. # of Words per bug report
AspectJ	03/02-01/14	593	1151	4.0	4068	188.5
Eclipse UI	10/01-01/14	6495	6228	2.7	4468	124.6
JDT	10/01-01/14	6274	5002	2.6	5233	137.4
SWT	02/02-01/14	4151	1415	2.1	5125	125.0
Tomcat	07/02-01/14	1056	1038	2.4	2320	87.5

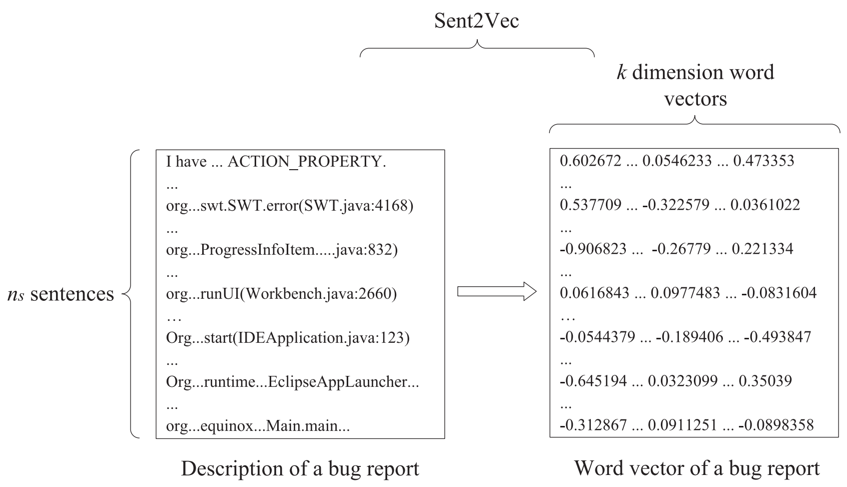


Fig. 6. An example of transforming descriptions in bug reports into vectors using Sent2Vec.

and k is the dimension of word embeddings. Sent2Vec is an extension of CBOW [29] to compose sentence embeddings. Unlike in CBOW, the context words running over the corpus in Sent2Vec are entire sentences rather than fixed-length context words, which meets the requirements of our model. For experimentation purpose, we have evaluated that the adopted strategy has a similar or even better performance with less memory and computation cost than other kinds of strategies (e.g., only word2vec).

Source files are composed of code tokens that are similar, but not identical, to natural languages [46]. Some keywords (e.g., int, public) are frequently presented in source code that may dampen the performance of Sent2Vec. In addition, it is difficult to represent source code in sentences. Thus, Sent2Vec is not good for converting source files into vectors. Importantly, the average word embeddings of words in a sentence have been proven to be efficient for obtaining sentence embeddings [15]. To dampen the effect of frequent keywords in the source code, this paper adopts weighted average word embeddings based on the revised TF-IDuF discussed in Section 2.1 to obtain the vectors of each line in source code as shown in the following equation:

$$V_l = \frac{1}{n_l}\sum_{i=1}^{n_l} w_i v_i = \frac{1}{n_l} wv \tag{9}$$

where in the lth line in a source file, $V_l \in \mathbb{R}^{1\times k}$ is the vector of the lth line of code, w_i ($w \in \mathbb{R}^{1\times n_l}$) is the rTF-IDuF term weight of the ith word in the lth line of code, and v_i ($v \in \mathbb{R}^{n_l\times k}$) is the word vector of the ith word in the lth line of code, in which n_l and k are the number of words in the lth line and the dimension of the word vector, respectively. It is better to use the right vectorized form during implementation, to accelerate the computational process. The word vector (v) is obtained using the word2vec technique with the Skip-gram model [29]. Some words may be absent in the set of pretrained words in word2vec. These words are initialized randomly and fine-tuned during training [17]. Ye et al. [49] verified that word embeddings trained on a Wiki corpus and a project-specific corpus (Eclipse and Java) had similar performance. However, a Wiki corpus has a greater vocabulary and more words (96 times and 548 times the size of a project-specific corpus, respectively), which is beneficial for deep learning models [54]. Thus, the pretrained word2vec used in our model is based on a Wiki corpus.

3.3. Feature detecting

After converting bug reports and source files into a set of vectors, the CNN model is applied to extract their features. Max-over-time pooling [18] is used when constructing a CNN. We first find the maximum number (n_S) of sentences from each bug report and the maximum number (n_L) of lines from each source file. They are then padded by zeros to be separately same with their maximum lengths [24]. Fig. 7 illustrates the process of feature detection from bug reports and source files using a CNN, in which k is the vector dimension of each sentence/line. The two CNNs have similar architecture, containing one convolutional layer with multiple filters followed by a max pooling layer [18]. We will evaluate how to choose the size of filters, the number of filters, and the number of convolutional layers when answering RQ1 in Section 4.3. The size of filters in the max pooling layer is $(n_S - \textit{the height of the filter size} + 1) \times 1$ for bug reports and $(n_L - \textit{the height of the filter size} + 1) \times 1$ for source files. The size of the feature vectors extracted from a bug report is *the number of filter sizes*the number of filters for each filter size*. The size of the feature vectors extracted from a source file is also *the number of filter sizes*the number of filters for each filter size*. We use different filter size in the max pooling layer to ensure that the final feature vectors extracted from a bug report and source file have the same size so that they can be concatenated in parallel as the input to the subsequent enhanced CNN.

3.4. Enhanced CNN training and prediction

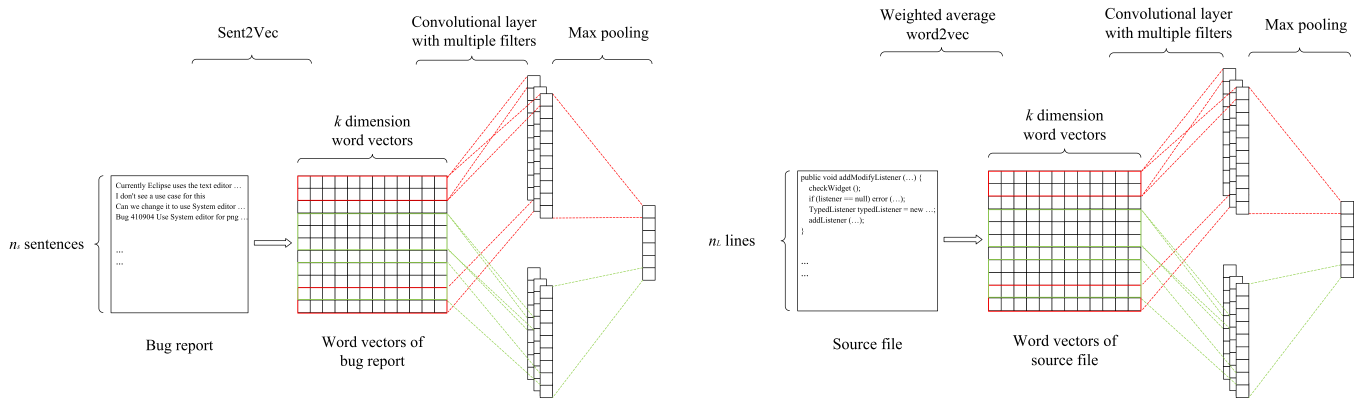
Although the conventional CNN has good performance in semantic parsing-related NLP problems [17], it cannot learn bug-fixing experience by itself. Therefore, this section discusses a novel technique to enhance the CNN for bug localization.

The conventional CNN is trained to minimize the following mean cross-entropy error function [10]:

$$\begin{aligned} L(\theta) &= \frac{1}{N}\sum_{i=1}^{N} cost_i \\ cost_i &= -\sum_{j=1}^{T} t_{ij} log(y_{ij}) \end{aligned} \tag{10}$$

where N is the number of samples in a batch and T is the number of classes. $cost_i$ is the cost function of sample i. t_{ij} is the true value of class j of sample i and y_{ij} is the output probability of class j of sample i.

Apart from the semantic information, the conventional CNN model contains no other important information related to bug localization, especially the fixing history of source files. Furthermore, bug-fixing recency and frequency have been verified as useful for bug localization by Lam et al. [22] and Ye et al. [48]. The change history of source files for bugs contains useful information for identification of fault-prone files [37]. This implies that source files fixed recently are more likely to be buggy than those fixed long ago (bug-fixing recency) and that source files that have been fixed many times are more likely to still contain bugs than those seldom fixed or even never fixed (bug-fixing frequency).



Sent2Vec
Convolutional layer with multiple filters
Max pooling
k dimension word vectors
n_s sentences
Bug report
Word vectors of bug report
Weighted average word2vec
Convolutional layer with multiple filters
Max pooling
k dimension word vectors
n_l lines
Source file
Word vectors of source file

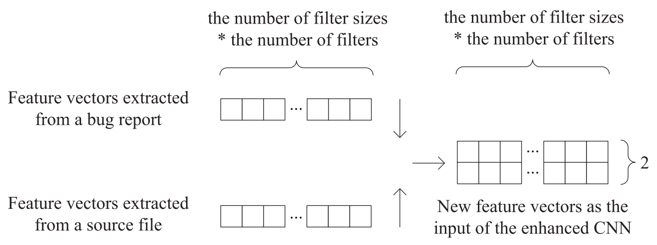
Currently Eclipse uses the text editor …
I don't see a use case for this
Can we change it to use System editor …
Bug 410904 Use System editor for png …
...
...

```
public void addModifyListener (…) {
    checkWidget ();
    if (listener == null) error (…);
    TypedListener typedListener = new …;
    addListener (…);
}
...
...
```

(a) Detecting features from bug reports

(b) Detecting features from source files

Fig. 7. The architecture of feature detection.



the number of filter sizes * the number of filters
the number of filter sizes * the number of filters
Feature vectors extracted from a bug report
Feature vectors extracted from a source file
2
New feature vectors as the input of the enhanced CNN

Fig. 8. The concatenation process.

We use equations similar to those in Ye et al.'s work [48] to identify the source file's bug-fixing recency and frequency. For a sample i with source file s and new bug report r, the bug-fixing recency is defined as follows:

$$R_i = \frac{1}{r.\ month - s.\ month + 1} \tag{11}$$

where *r.month* represents the month in which bug report *r* was generated and *s.month* denotes the month in which source file *s* was last fixed before bug report *r* was created. *r.month* and *s.month* take the year into account. For example, if *r.month* is January this year and *s.month* is December last year, R_i is 0.5. Therefore, the bug-fixing recency R_i is the inverse of the time interval between the creation of the bug report and the last time the source file was fixed.

The bug-fixing frequency F_i is expressed by the number of times source file s was fixed before bug report r was submitted [48].

R_i and F_i may have wide value ranges that make them incomparable, which may result in detrimental effects [48]. In both the training and testing sets, R_i and F_i are scaled as follows:

$$r_i = \begin{cases} 0 & if\ R_i < R_{min} \\ \dfrac{R_i - R_{min}}{R_{max} - R_{min}} & if\ R_{min} \le R_i \le R_{max} \\ 1 & if\ R_i > R_{max} \end{cases}$$

$$f_i = \begin{cases} 0 & if\ F_i < F_{min} \\ \dfrac{F_i - F_{min}}{F_{max} - F_{min}} & if\ F_{min} \le F_i \le F_{max} \\ 1 & if\ F_i > F_{max} \end{cases} \tag{12}$$

where $R_{\min}$ ($F_{\min}$) and $R_{\max}$ ($F_{\max}$) are the minimum and maximum values of R_i (F_i) in the training set. During testing, R_i (F_i) may be larger than $R_{\max}$ ($F_{\max}$) or smaller than $R_{\min}$ ($F_{\min}$) obtained in the training phase, so the first and last judgments are necessary.

To improve the bug localization performance, we enhance the conventional CNN by adding bug-fixing recency and frequency to the cost function in the fully connected layer as two penalty terms. The new cost function that considers bug-fixing recency (r_i) and bug-fixing frequency (f_i) is defined as follows:

$$cost_i = -\sum_{j=1}^{T} t_{ij} log\,(y_{ij}) - \omega_1 r_i - \omega_2 f_i \tag{13}$$

where ω_1 and ω_2 are initialized by random values from a truncated normal distribution and tuned in the training phase similarly to other weights. r_i and f_i are the scaled values of bug-fixing recency and frequency, respectively.

Now we use the enhanced CNN to learn the relationship between bug reports and corresponding buggy files. The feature vectors ($\in \mathbb{R}^{1\times(the\ number\ of\ filter\ sizes\ *\ the\ number\ of\ filters\ for\ each\ filter\ size)}$) extracted from bug reports and those ($\in \mathbb{R}^{1\times(the\ number\ of\ filter\ sizes\ *\ the\ number\ of\ filters\ for\ each\ filter\ size)}$) from source files are concatenated in parallel into 2-dimensional feature vectors ($\in \mathbb{R}^{2\times(the\ number\ of\ filter\ sizes\ *\ the\ number\ of\ filters\ for\ each\ filter\ size)}$) as the input to the enhanced CNN. The concatenation process is shown in Fig. 8. The convolutional layer in the enhanced CNN uses filters with the size of 2 × 2 to learn the correlated relationships between bug reports and source files by convolving the input of the enhanced CNN that is a concatenation of the feature vectors extracted from bug reports and source files. The size of filters in the max pooling layer is 2 × (*the number of filter sizes * the number of filters for each filter size*). The enhanced CNN was used to correlate bug reports to buggy files including both linear and nonlinear relationships because of its convolutional operation and fully connected layer. Other factors (bug-fixing recency and frequency) are included in the enhanced CNN using Eq. (13).

All three CNNs use max-over-time pooling and are regularized by batch normalization to speed up the training process. DeepLoc is trained using the Adam (Adaptive Moment Estimation) Optimizer [19], which uses an adaptive learning algorithm. Compared to Adadelta [50], which retains an exponentially decaying average of past squared gradients, Adam can also store the exponentially decaying average of past gradients. In addition, Adam has shown better performance in large empirical investigations than other algorithms that automatically adjust learning rates [19].

After training, the bug reports are related to buggy files rather than matched by textual similarity. When a new bug report appears, it is paired with each source file and then input into the trained DeepLoc to predict buggy files.

4. Experiments

Several experiments are conducted to evaluate DeepLoc's performance. We describe the datasets used, the experimental setup and the performance metrics in Section 4.1. The research questions are listed in Section 4.2, and the results in Section 4.3.

4.1. Experimental settings and evaluation metrics

Because some source files may have been modified or even deleted, we may fail to find the original mapped source files for an old bug without before-fix versions. Thus, we collect a dataset (Table 1) with before-fix versions of source files and relate them to each bug based on publicly available mappings of bug reports and the corresponding commit history provided by [48].

The chronologically sorted bug reports for each project are split into 60% as the training set (oldest bugs), 20% as the validation/development set, and 20% as the test set (newest bugs). Large numbers of source files exist for a given bug report, and it is infeasible to involve them all in the training set because of the necessary training time and memory. For fair comparison with other state-of-the-art techniques, we use a procedure similar to that in [22,48]. In training, for each bug report, we rank the cosine similarity of the word vectors between the bug report and all of the source files and then select the top 300 irrelevant files as the negative samples, which implies that the actual buggy files and other 300 least similar files are together with each bug report to be set as the training set. In the validation and testing set, each bug report is paired with all source files. We use tensorflow to construct the model, and all experiments are run on a server with CPU Intel Xeon CPU E5-4620 2.20 GHz (32 cores), 128 GB RAM.

To evaluate DeepLoc's performance, three metrics are adopted: Accuracy@k, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP). These metrics have been widely used in existing bug localization studies [22,48,49,53]. Their definitions are expressed as follows.

- *Accuracy@k* measures the percentage of bug reports for which at least one real buggy file is located within the top *k* rank.
- *MRR* is the mean of the Reciprocal Rank that accumulates the inverse of the position of the first correctly-located buggy file for each bug. For a set of *Q* bug reports, the MRR in this paper is computed as follows:

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{first_i} \tag{14}$$

  where $first_i$ denotes the position of the first correctly-located buggy file for the *i*th bug report.
- *MAP* can measure performance when a query has multiple relevant documents. It is more suitable in the field of bug localization because a bug report may have more than two buggy files on average (as shown in Table 1). MAP is the mean of the Average Precision (AvgP) values of *Q* bug reports, each of which is the average of the precision values for a bug report. The formulation of MAP for bug localization is as follows:

$$MAP = \frac{1}{Q}\sum_{i=1}^{Q}AvgP(i) \tag{15}$$

$$AvgP(i) = \sum_{j=1}^{M}\frac{(T(j)/j)\times ind(j)}{B(i)} \tag{16}$$

  where *M* is the maximum position of correct buggy files for the *i*th bug report located using the bug localization technique, $ind(j)$ indicates whether the file located in rank *j* is the correct buggy file ($ind(j) = 1$) or not ($ind(j) = 0$), $B(i)$ stands for the number of buggy files for the *i*th bug report, and $T(j)$ denotes the number of buggy files in the top *j*.

The higher the Accuracy@k, MRR, and MAP values, the better the performance of the bug localization technique.

4.2. Research questions

Specifically, we aim to answer the following questions.

RQ1: What effect do the model settings have on DeepLoc? Before building DeepLoc, we need to analyze three settings using the validation set: the size of filters, the number of filters, and the number of convolutional layers.

RQ2: What effect does the enhanced CNN have on DeepLoc? We then evaluate whether the enhanced CNN improves DeepLoc's accuracy against the conventional CNN. The only difference between the two CNNs is that the enhanced CNN considers bug-fixing recency and frequency, but the conventional CNN does not. Furthermore, we examine their time overheads for prediction.

RQ3: How good is DeepLoc's performance compared to state-of-the-art techniques? To validate DeepLoc's effectiveness and efficiency, four state-of-the-art approaches are used as competitors:

- *DeepLocator* proposed in our previous work [47] uses CNN to extract features from bug reports and source files that are pre-treated using rTF-IDuF and AST.
- *HyLoc* proposed by Lam et al. [22] combines deep learning with information retrieval (IR) technique.
- *LR + WE* proposed by Ye et al. [49] enhances the previously proposed learning-to-rank (LR) model [48] with word embedding (WE).
- *BugLocator* proposed by Zhou et al. [53] uses a revised Vector Space Model (rVSM) to measure the textural similarity between bug reports and source files.

Both **RQ2** and **RQ3** use a testing set for evaluation. There are 50 instances in each batch.

4.3. Results and analyses

RQ1: Model performance under different model settings.

We first evaluate how to choose the size of filters using the validation set. The number of filters is set as constant (100) as a control variable. Fig. 9 reports the MAP values of each dataset with the filter size ranging from $1 \times k$ to $10 \times k$ (where *k* is the width of word vectors). The results show that the best filter size for the five datasets is $3 \times k$ (AspectJ), $6 \times k$ (Eclipse UI), $4 \times k$ (JDT), $3 \times k$ (SWT), and $4 \times k$ (Tomcat) respectively. According to the empirical analysis of Zhang and Wallace [52], using multiple filter sizes near the best single size produces the best results. So in our experiments, the filter sizes are set as ($2 \times k$, $3 \times k$, $4 \times k$), ($5 \times k$, $6 \times k$, $7 \times k$), ($3 \times k$, $4 \times k$, $5 \times k$), ($2 \times k$, $3 \times k$, $4 \times k$), and ($3 \times k$, $4 \times k$, $5 \times k$) for the five datasets

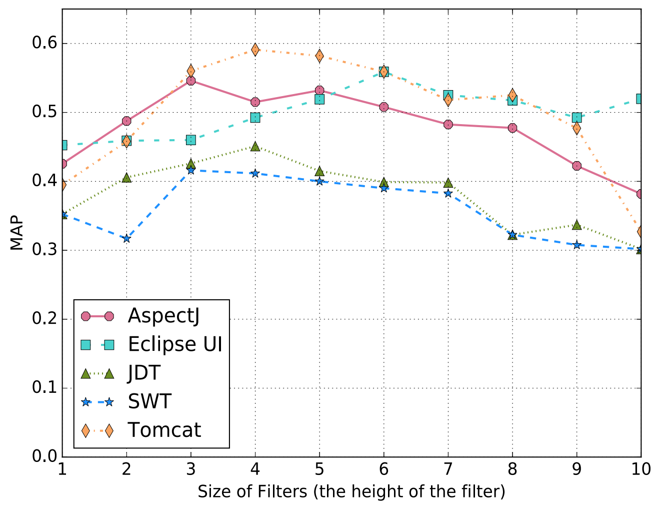


Fig. 9. Performance of different filter sizes.

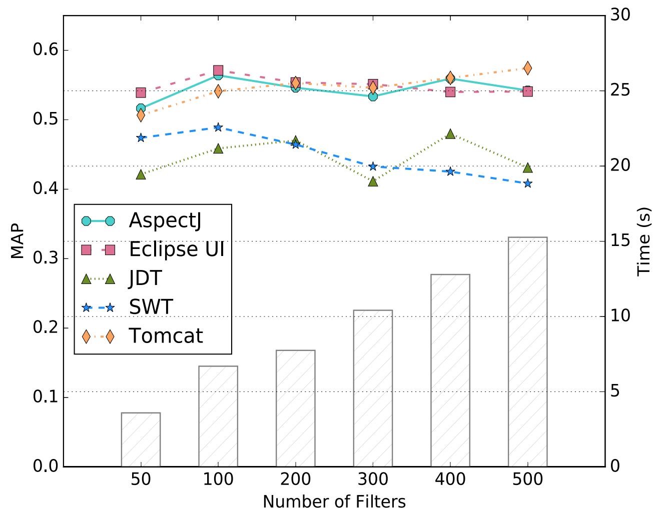


Fig. 10. Performance of number of filters.

(AspectJ, Eclipse UI, JDT, SWT, and Tomcat). We fine-tune the number of filters for each dataset given the formerly determined filter size. We consider values {50, 100, 200, 300, 400, 500}. Fig. 10 represents the MAP values (line chart) and the number of filters for the five datasets. We also provide the average training time per batch (bar chart) for the five datasets to help us choose the appropriate number of filters balanced between the results and the time consumption. The results show that the greater the number of filters, the higher the computation time. However, the MAP values do not always increase because more filters may lead to overfitting. After weighing the trade-offs, we set the number of filters as 100 for the whole dataset.

Next, we compare the MAP values and the average time consumption for each batch of the CNN, using one convolutional layer and two convolutional layers. When using a two-layer CNN, the features between words are convolved again, which is able to extract deeper semantic information. Table 2 demonstrates that the performance of the two-layer CNN is slightly better than that of the one-layer CNN, but the time consumption is much higher. Thus, DeepLoc uses a one-layer CNN.

RQ2: The enhanced CNN's effects on DeepLoc.

In this section, we use the testing set to compare DeepLoc's performance using enhanced CNN and conventional CNN. We find that both the MAP and MRR of the enhanced CNN are higher than those of the conventional CNN, with a 10.87%–13.4% higher MAP (as shown in Fig. 11). Because CNN does not have long-term memory, it cannot learn the features of bug-fixing recency and frequency, which are included in the enhanced CNN.

We report the time overhead of DeepLoc using the enhanced CNN and the conventional CNN. Because more weights in DeepLoc must be adjusted in the training phase than in the CNN, DeepLoc's average time consumption per batch in training is slightly higher (about 0.03 seconds) than the CNN's. Table 3 presents the average prediction time for one bug report by DeepLoc using the enhanced CNN and the conventional CNN, and the existing deep learning-related model (HyLoc) [22].

Table 2
MAP and time consumption for different numbers of layers.

Project	One-layer		Two-layer	
	MAP	Time (s)	MAP	Time (s)
AspectJ	0.55	6.36	0.58	212.18
Eclipse UI	0.56	6.92	0.57	217.15
JDT	0.48	6.38	0.49	212.86
SWT	0.49	6.65	0.48	213.56
Tomcat	0.57	6.74	0.59	213.91

The first two rows demonstrate that DeepLoc's average prediction times using the enhanced CNN and the conventional CNN are very similar.

RQ3: DeepLoc's performance.

Fig. 12 presents the Accuracy@k results for the four models (DeepLoc, HyLoc, LR + WE, and BugLocator) on Project SWT, with k ranging from 1 to 20. The results show that BugLocator has the worst performance. The difference between HyLoc and LR + WE is not clear. DeepLoc shows evident improvements, especially from Top 1 to 11, which is more significant for bug localization because developers prefer to search only a few source files to find buggy files. For example, DeepLoc achieves Accuracy@k of 39.0%, 65.7%, and 77.1% for $k = 1, 5, 10$, respectively. That is to say, it can correctly locate buggy files for 39.0% of bug reports when recommending only one source file to developers, and it can fix 77.1% of bug reports when ten source files are recommended. In comparison, HyLoc achieves Accuracy@k 33.4%, 71.1%, LR + WE achieves 34.0%, 71.0%, and BugLocator achieves 22.3%, 51.7% for $k = 1, 10$, respectively. Using a Mann-Whitney-U-Test [26], the Accuracy@k results of HyLoc and LR + WE are not significantly different, but DeepLoc significantly ($p < 0.05$) outperforms both of them.

Table 4 lists the Accuracy@1,5,10, MAP, and MRR values of the five models using the test set for five projects. Again, DeepLoc outperforms the other four models for all five projects.

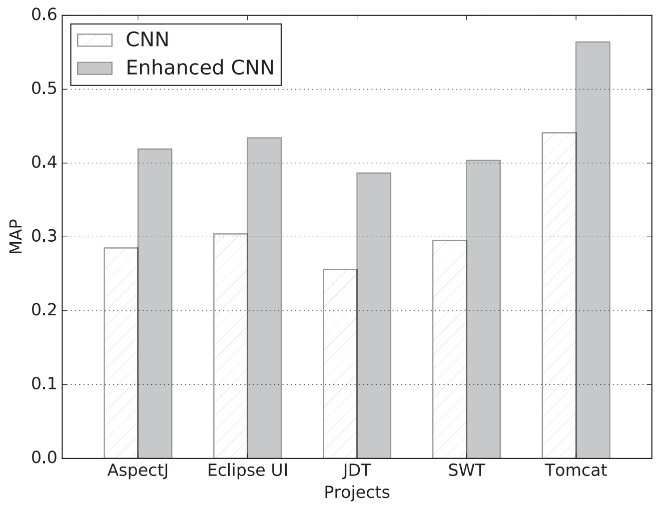
Compared to DeepLocator, DeepLoc achieves 2%–5% higher Accuracy@1 and 1%–5% higher Accuracy@10. DeepLoc distinguishes bug reports and source files using two different word-embedding techniques and CNNs. All information in bug reports and source files is included and deeper networks are used to further improve the bug localization performance.

In addition, DeepLoc achieves 16%–24% greater Accuracy@1 than BugLocator. Unlike BugLocator, whose performance for unseen bug reports is unclear due to the only parameter α being tuned on the same dataset for both training and evaluation [53], all of the weights for DeepLoc are learned on the training set, tuned on the validation set, and then used to predict the testing set, whose performance is scalable. At the same time, LR + WE achieves 2%–16% lower Accuracy@1 than DeepLoc. LR + WE may be limited by the linear weighted sum of features, but DeepLoc does not suffer this problem because of the convolutional networks.

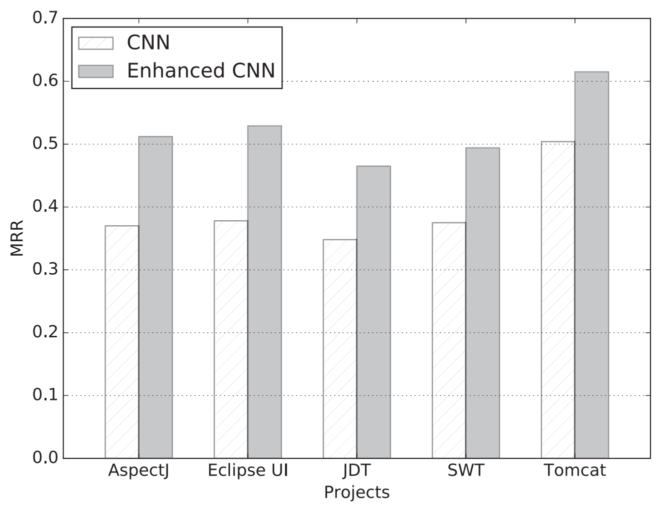
Finally, DeepLoc achieves 2%–10% higher Accuracy@1 than HyLoc. Because Lam et al. [22] validated HyLoc's comparable or superior performance against existing approaches [34,48,49,53], we use HyLoc as the main competitor and analyze the results. When analyzing the ranking results, we observe that some buggy files given low scores by HyLoc can be located more precisely by DeepLoc. For example, HyLoc gives one buggy file for Bug 54450 in Project Tomcat Rank 50, which is a relatively low score. However, DeepLoc predicts it as a buggy file by ranking it in fourth place because "resource" in bug reports is paired with words related to "context" (getContext, configureContext, processContextConfig) in source files many times. Another example is Bug 399401 in Project Eclipse UI. The rank for its buggy file given by HyLoc is 38. However, "perspective" is paired with words related to "view" and "visible" (getViewLayout, isPartVisible), so DeepLoc can relate them if these pairs have ever appeared in older bug reports and corresponding buggy files used in its training. Therefore, even when tokens in source files are mismatched with the words used in bug reports, DeepLoc can relate them, which indicates that DeepLoc can learn correlations if they frequently appear as pairs. In particular, DeepLoc's prediction time, shown in Table 3, for one bug report is much lower than HyLoc's, which indicates that DeepLoc is more suitable in practice.

5. Discussion

The major challenge for bug localization is the semantic gap between bug reports and source code files. Textual similarity, which is used in most existing techniques [16,22,48,53], is based on the term



(a) MAP



(b) MRR

Fig. 11. MAP and MRR of conventional CNN and enhanced CNN for five projects.

Table 3
Prediction time (seconds) per bug report for three models.

Prediction time	AspectJ	Eclipse UI	JDT	SWT	Tomcat
DeepLoc	127.9	93.5	152.0	51.0	36.5
CNN	123.3	95.3	152.3	53.2	36.3
HyLoc	144.0	126.0	198.0	108.0	60.0

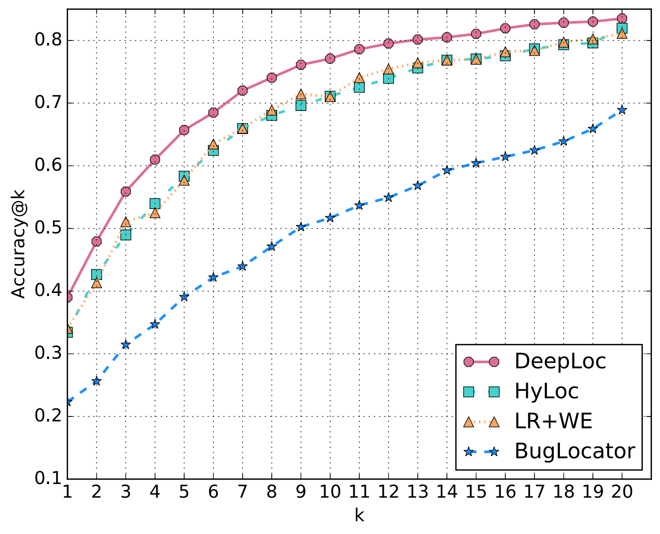


Fig. 12. Accuracy@k of the four models on Project SWT.

frequency rather than semantic information of the words and phrases. Unlike textual similarity, DeepLoc correlates bug reports to the corresponding buggy files based on a deep understanding of semantics.

DeepLoc uses word-embedding techniques to map bug reports and source files into semantic vector spaces where similar words are close to each other. According to their characteristics, DeepLoc uses two different word embedding-based techniques for bug reports and source files. Fig. 13 visualizes the learned embeddings of 500 bug reports from Project SWT using t-SNE [40], where a number represents a bug report. Each bug report is transformed into a word vector as discussed in Section 3.2. These vectors are then projected into 2-D dimension by t-SNE to make the vectors visible as shown in Fig. 13. The closer two bug reports are in the vector space, the greater the similarity between them. In Example I, the three bug reports are “Unused and breaks compilation on GTK 3.x. Bug 391413 Remove GdkGCValues_sizeof”, “Remove GdkVisual_sizeof - not used at all and breaks compilation on GTK 3.x. Bug 391408 Remove GdkVisual_sizeof”, and“Method is not used anywhere and breaks compilation on GTK 3.x. Bug 391404 Remove GdkImage_sizeof”. They are all related to GTK and the “sizeof” operation. Most importantly, these three bug reports point to a same buggy file, namely OS.java, but the first bug report still contains other buggy files. The contents of the bug reports in Example II, ”Bug 369228 Kill pre GTK 2.4 leftovers from Tree”, “Bug 369227 Kill pre GTK 2.4 leftovers from List”, “Bug 369226 Cleanup pre GTK 2.4 leftovers in Table”, and “Bug 369225 Cleanup pre GTK 2.4 leftovers”, also share similar operations. In addition, the first three bug reports share the same buggy files. The bug reports in Example III also share the same or similar words like “widget”, “unused 2.10 version guard” and so on. The aggregation of some points in Fig. 13 indicates that these bug reports may share similar meanings or even point to the same buggy files. Therefore, the word-embedding technique is useful for converting text into vectors that retain semantic information, and can help to localize buggy files. However, it is still insufficient, which is why it requires consequent models to detect features and learn correlations between bug reports and source files.

The embedded vectors that contain information about semantic similarities greatly benefit DeepLoc. The CNN model performs well in the semantic parsing field because of the convolving filters [17], which helps DeepLoc to extract hidden semantic information from bug reports and source files. DeepLoc applies two CNNs to extract features from bug reports and source files and then uses an enhanced CNN to correlate bug reports and buggy files including both linear and nonlinear relationships, which makes it different from the studies in [48,49], which use a linear weighted sum of features. Bug-fixing history is also included in DeepLoc, to make it more powerful.

6. Threats to validity

The experimental results demonstrate DeepLoc’s feasibility, however, we acknowledge some potential threats to the validity of our approach and experiments. Following the suggestions of Wohlin et al. [44], we discuss threats to internal validity, construct validity, external validity, and statistical conclusion validity.

6.1. Internal validity

The proposed approach converts each sentence in a bug report into

Table 4
Performance comparison.

Project	Model	Accuracy@1	Accuracy@5	Accuracy@10	MAP	MRR
AspectJ	DeepLoc	0.45	0.71	0.80	0.42	0.51
	DeepLocator	0.40	0.66	0.78	0.34	0.49
	HyLoc	0.38	0.65	0.75	0.32	0.48
	LR + WE	0.29	0.58	0.74	0.30	0.45
	BugLocator	0.22	0.46	0.58	0.28	0.36
Eclipse UI	DeepLoc	0.45	0.70	0.79	0.43	0.53
	DeepLocator	0.43	0.66	0.74	0.42	0.51
	HyLoc	0.40	0.64	0.73	0.41	0.51
	LR + WE	0.39	0.60	0.71	0.40	0.46
	BugLocator	0.29	0.50	0.60	0.33	0.38
JDT	DeepLoc	0.43	0.65	0.77	0.44	0.53
	DeepLocator	0.40	0.64	0.73	0.39	0.47
	HyLoc	0.33	0.59	0.69	0.34	0.45
	LR + WE	0.41	0.65	0.75	0.42	0.52
	BugLocator	0.19	0.40	0.51	0.29	0.37
SWT	DeepLoc	0.39	0.66	0.77	0.40	0.49
	DeepLocator	0.36	0.60	0.75	0.39	0.48
	HyLoc	0.33	0.58	0.71	0.37	0.45
	LR + WE	0.34	0.57	0.71	0.38	0.45
	BugLocator	0.22	0.39	0.52	0.27	0.31
Tomcat	DeepLoc	0.54	0.72	0.81	0.56	0.62
	DeepLocator	0.52	0.72	0.80	0.54	0.60
	HyLoc	0.52	0.71	0.78	0.52	0.60
	LR + WE	0.49	0.70	0.76	0.50	0.56
	BugLocator	0.36	0.62	0.71	0.43	0.48

a vector, which could be affected by stemming and removal of stop words process. This potential will be investigated in a future study. Both bug reports and source files are transformed into vectors based on word-embedding techniques. These techniques make texts from bug reports and source files into adequate input for CNNs, retaining their semantic information and saving memory space. However, the performance of the proposed approach relies to some extent on the ability of word-embedding techniques. It would be best to test these techniques before adding to the proposed model. Improving these techniques will also help to enhance our model. We leave this for future studies. Finally, the analysis on the choice of filter size (RQ1) is related to the writing style of the developers. For example, if the developers in a project team prefer to use very long phrases to express bug reports, the filter size should also be long. We analyze the results of the dataset and provide a general conclusion, which may not be adequate.

6.2. *Construct validity*

As shown in Table 1, the dataset for bug localization is imbalanced. Because the focus of this paper is not a data imbalance problem, we selected a simple sampling strategy. A better technique for solving a data imbalance is important for bug localization, which will be investigated in detail in a future study. On the other hand, we split the dataset into a training set (60%), a validation set (20%), and a testing set (20%). This approach is adopted in most machine learning-related papers [6,32,36] but not in bug localization-related papers [16,48,53]. The effects of various splitting strategies on deep learning-related bug localization techniques is still unknown and is worthy of investigation in further studies.

6.3. *External validity*

Because the codes for HyLoc, LR + WE, and BugLocator are not published, we implemented them according to the algorithms provided by Lam and Co-workers [22,49,53] and obtained similar results. However, the results are not identical. Fortunately, Ye et al. [48] and Lam et al. [22] provided their results for the same datasets. We choose the best results for each project comparing our results to theirs. Finally, the datasets used were obtained from Java projects. The results may not be generalizable to other projects written in other programming languages. In the future, we intend to improve the model for use in other projects written in different programming languages.

6.4. *Statistical conclusion validity*

In this study, we used a Mann–Whitney-U-Test for the Accuracy@k analyses. We provided Accuracy@k results of the four models (DeepLoc, HyLoc, LR + WE, BugLocator) with k ranging from 1 to 20 and applied a Mann-Whitney-U-Test for comparison. However, the sample size is close to the minimum required number of observations in a standard statistical test. We thus admit the experimental results might be subject to a threat to validity.

7. Related work

7.1. *Our previous work*

In [47], Xiao et al. proposed DeepLocator for improving the performance of bug localization. DeepLocator applied Abstract Syntax Tree (AST) to represent syntax and extract the programming patterns of source code. Word2vec was then used to map the preprocessed words of both bug reports and source files into semantic vector spaces in which similar words are near each other. However, there were always many words in bug reports and source files even though they were pretreated using revised TF-IDuF and AST. Therefore, a large amount of memory space was needed to store the word vectors. In addition, the preprocessing procedure also caused the loss of some important information, which limited DeepLocator's performance. When training, only one CNN was used to extract features from the word vectors of both bug reports and source files and classify buggy files. The use of the same CNN to deal with bug reports and source files ignored the differences between them and thus dampened DeepLocator's accuracy. To solve the aforementioned problems, this paper proposes DeepLoc, which uses two different word-embedding techniques and CNNs to distinguish bug reports and source files, to further improve bug localization.

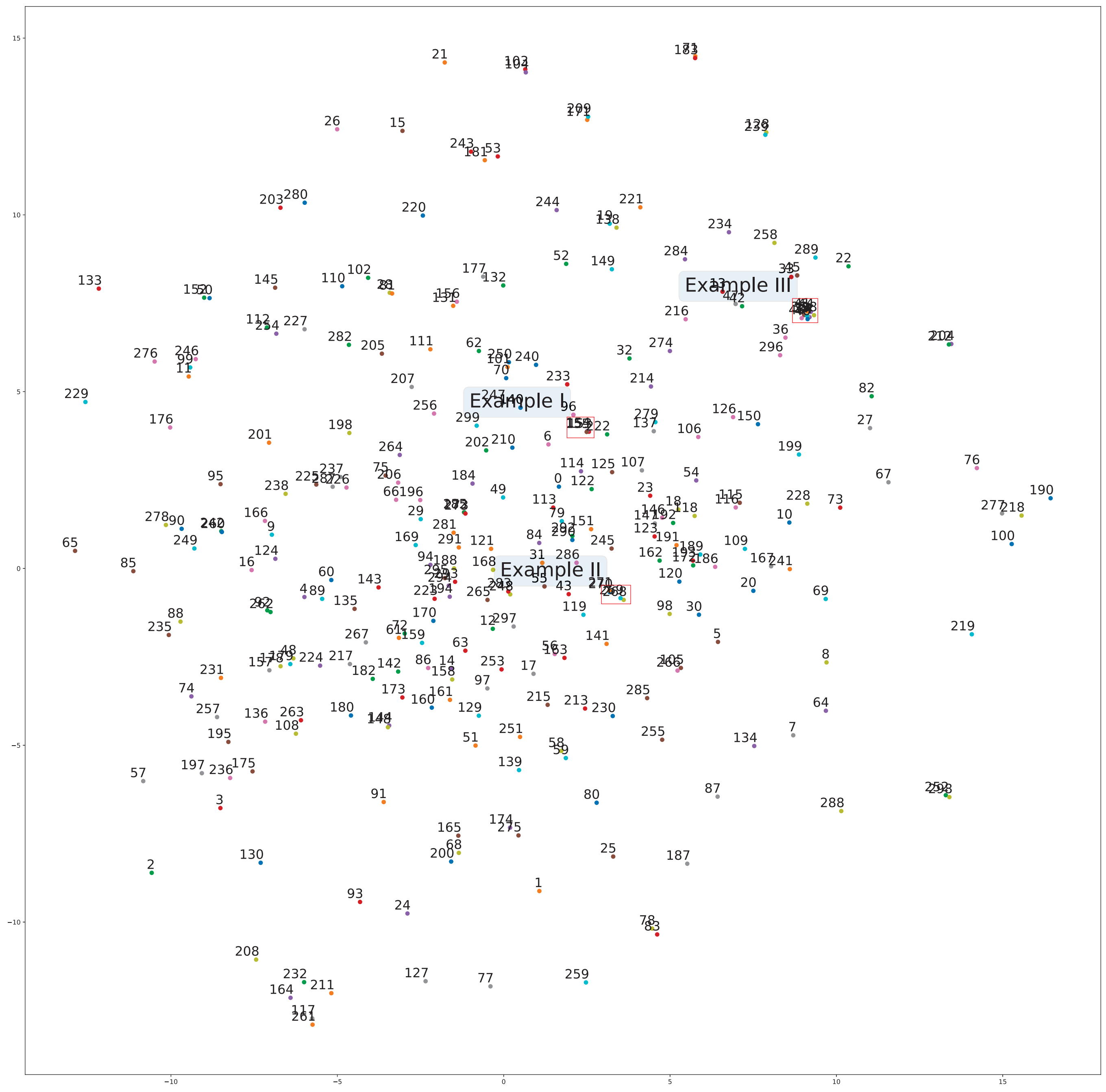


Fig. 13. Visualization of the learned embeddings of 500 bug reports from Project SWT using t-SNE.

7.2. Bug localization

In the literature, several techniques have been proposed to help localize bugs. The commonly used techniques are either based on information retrieval (IR) methods, machine learning (ML) methods, or deep learning (DL) methods.

IR: Zhou et al. [53] proposed BugLocator, which was based on a revised Vector Space Model (rVSM), to measure the textural similarity between bug reports and source files using information from similar bugs that had previously been fixed. The weighted sum of the two rankings was used to locate the relevant files for a bug. BugLocator was a famous technique that attempted to rank relevant source files for bug reports. Wong et al. [45] proposed segmentation and stack-trace analysis techniques on top of BugLocator [53], to boost fault localization. Saha et al. [38] regarded the summary and description of bug reports as two different query fields, combined respectively with classes, methods, variables, and comments extracted from source files. However, these eight features were considered equally important. Moreover, they used the fixed version of the project in the evaluation, which led to inconsistency when considering future bug-fixing information.

To address the issues mentioned above, Ye et al. [48] developed an adaptive ranking model supported by a learning-to-rank (LR) technique to rank relevant files for bug reports. Six features were used in this paper: surface lexical similarity, application programming interface (API)-enriched lexical similarity, collaborative filtering score, class name similarity, bug-fixing recency, and bug-fixing frequency. In the literature [49], LR + WE was proposed to enhance LR, in which bug reports and source code were represented by word embeddings and the similarities between them were added as additional features. The weight of each feature was then trained using the LR technique based on the previously fixed bugs. However, this model used a hill-climbing algorithm with a linear weighted sum of features, which might ignore nonlinear relationships. These IR-based techniques rely on the textual similarities between bug reports and source files, which do not bridge the semantic gap.

ML: Kim et al. [16] applied a Naive Bayes classifier to build a two-phase recommendation model. Phase 1 was used to decide whether the bug was predictable or not. If not, there was no further prediction behavior. If it was predictable, Phase 2 applied a Bayes model to recommend a set of files to fix. However, the model mainly focused on the names of the fixed files. Thus, it was difficult for the model to recommend a file that had never been fixed before. Moser et al. [31] built a model based on three common machine learners to find defective files from Eclipse projects considering code changes. Markov logic was used in [51] by combining statement coverage, static program structure information, and prior bug information. The authors, however, did not consider contextual information (failure explanations). BugSout, a topic-based machine learning model proposed by Nguyen et al. [34], used an extended Latent Dirichlet Allocation (LDA) [4]. The bugs were related to their corresponding buggy files by their shared topics. However, a tuning process was needed to find the right number of topics for different projects, which meant that the model was not automated. In addition, there was a strong assumption in this paper that the textual contents of bug reports shared some technical aspects with the textual contents of the corresponding buggy files. However, natural language texts used in bug reports likely differ from code tokens in the source code files of real-world projects, which implies that this assumption is insufficient.

DL: Deep learning is a relatively new concept, and there is little prior work about deep learning-based bug localization. Lam et al. [22] built a model named HyLoc that combined deep learning with the IR technique. About six deep neural networks (DNNs) were used in this model: two for feature extraction, two for projection, one for relevancy estimation, and one for feature combination. However, using so many kinds of DNNs posed a risk of complex weight adjustment and high training and learning costs. Their experiments showed that using DNNs alone achieved poor performance and that most improvements still benefited from IR techniques, which implied that DNNs in their model was a subsidiary.

Previous studies have proposed several features to represent program source code. However, few methods try to extract semantic information from source code. Deep learning can help to bridge this gap.

7.3. Deep learning in software engineering

In recent years, many deep learning techniques have been introduced to software engineering research, such as code suggestion, API suggestion, defect prediction, effort estimation, program classification, and bug localization.

White et al. [43] proposed a deep architecture to model software language specific to sequential data and suggested many avenues for future work, especially for code suggestion. A novel tree-based convolutional neural network (TBCNN) was built by Mou et al. [32] for programming language processing. Wang et al. [42] used a deep belief network to detect features from tokens extracted from an AST of source code to find defective files, but it was insufficient for mapping the tokens to continuous integers.

Choetkiertikul et al. [6] proposed a deep learning-based prediction system for estimating story points based on Long Short-Term Memory (to learn a vector representation for issue reports) and Recurrent Highway Network (to build a deep representation). DeepAPI was proposed by Gu et al. [10] to find API usage sequences given an API-related natural language query. The model adopted the attention-based recurrent neural network (RNN) Encoder-Decoder model considering API importance by using IDF-based weighting to train API usage sequences and their corresponding annotations. The training time of this model was very long, which is also a drawback of RNN.

In the software engineering field, many files are written in natural languages, such as bug reports, API descriptions, and annotations. Even source codes themselves are similar to natural language to some extent. Thus, many mature techniques used in natural language processing can be applied to some issues in software engineering to extract semantic information.

8. Conclusions and future work

The existing approaches to bug localization focus on the similarities or relationships between the term weights of words from bug reports and source files. However, most of these approaches ignore the semantic information hidden in the bug reports and source files. In particular, few approaches consider the semantics of an entire source code. DeepLoc, a deep learning-based model that consists of an enhanced CNN, together with word-embedding techniques and two CNNs for detecting features from bug reports and source files, is proposed to address these challenges and improve the performance of bug localization. Unlike the existing approaches, word-embedding techniques are adapted to convert each sentence of bug reports and each line of source code into vectors, retaining their semantics in vector space. These vectors are then fed into the CNNs to extract their hidden semantics and features. An enhanced CNN is used to detect the correlation between the feature vectors extracted from bug reports and source code considering the bug-fixing experience (bug-fixing recency and frequency).

We provide empirical suggestions on how to construct the model and evaluate DeepLoc's performance against four state-of-the-art bug localization techniques (DeepLocator, HyLoc, LR + WE, and BugLocator) using more than 18,500 bug reports extracted from five projects. The experimental results show that DeepLoc performed better than conventional CNN (10.87%–13.4% MAP improvements), which indicates the importance of bug-fixing experience for bug localization. Compared to the other four approaches, DeepLoc achieved higher Accuracy@k, MAP, and MRR, using less computation time. DeepLoc bridges the semantic gap by adapting word embedding to retain the semantics of bug reports and source code, CNNs to parse their syntax, and the enhanced CNN to learn the correlations between bug reports and source code considering bug-fixing experience.

However, the existing learning-based models, including our model, seldom consider the orders of buggy files during training, which may limit their performance when using metrics such as MAP and MRR to test their ability. In the future, we intend to improve DeepLoc's sensitivity to buggy file orders. We will also investigate DeepLoc's performance using other TF-IDF weight schemes [27,48]. Few papers have studied the effect of imbalanced data on bug localization and relevant techniques, which we leave for future work.

Acknowledgments

This work is supported in part by. the General Research Fund of the Research Grants Council of Hong Kong [No. 11208017], and the research funds of City University of Hong Kong [No. 7004683]

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, A.J. Van Gemund, A practical evaluation of spectrum-based fault localization, J. Syst. Softw. 82 (11) (2009) 1780–1792.
[2] J. Beel, S. Langer, B. Gipp, TF-IDuF: a novel term-weighting scheme for user modeling based on users' personal document collections, Proceedings of the iConference 2017, Wuhan, China, (2017). URL http://ischools.org/the-iconference/ .
[3] D. Binkley, M. Davis, D. Lawrie, C. Morrell, To camelcase or under_score, Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, IEEE, 2009, pp. 158–167.
[4] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent Dirichlet allocation, J. Mach. Learn. Res. 3 (Jan) (2003) 993–1022.
[5] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns

and Java-(Required), Prentice Hall, 2004.
[6] M. Choetkiertikul, H.K. Dam, T. Tran, T.T.M. Pham, A. Ghose, T. Menzies, A deep learning model for estimating story points, IEEE Transactions on Software Engineering (2018).
[7] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (Aug) (2011) 2493–2537.
[8] W.B. Croft, D. Metzler, T. Strohman, Search Engines: Information Retrieval in Practice, vol. 283, Addison-Wesley, Reading, 2010.
[9] K. Fukushima, S. Miyake, Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition, Competition and Cooperation in Neural Nets, Springer, 1982, pp. 267–285.
[10] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 631–642.
[11] A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, Software Engineering (ICSE), 2012 34th International Conference on, IEEE, 2012, pp. 837–847.
[12] D.H. Hubel, T.N. Wiesel, Receptive fields, binocular interaction and functional architecture in the cat's visual cortex, J. Physiol. 160 (1) (1962) 106–154.
[13] S. Ioffe, C. Szegedy, Batch normalization: accelerating deep network training by reducing internal covariate shift, International Conference on Machine Learning, (2015), pp. 448–456.
[14] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2005, pp. 273–282.
[15] T. Kenter, A. Borisov, M. de Rijke, Siamese cbow: optimizing word embeddings for sentence representations, arXiv:1606.04640 (2016).
[16] D. Kim, Y. Tao, S. Kim, A. Zeller, Where should we fix this bug? A two-phase recommendation model, IEEE Trans. Software Eng. 39 (11) (2013) 1597–1610.
[17] Y. Kim, Convolutional neural networks for sentence classification, arXiv:1408.5882 (2014).
[18] Y. Kim, Y. Jernite, D. Sontag, A.M. Rush, Character-aware neural language models. AAAI, (2016), pp. 2741–2749.
[19] D. Kingma, J. Ba, Adam: a method for stochastic optimization, arXiv:1412.6980 (2014).
[20] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Advances in Neural Information Processing Systems, (2012), pp. 1097–1105.
[21] B. Kwolek, Face detection using convolutional neural networks and gabor filters, Artificial Neural Networks: Biological Inspirations–ICANN 2005, (2005), pp. 551–556.
[22] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Combining deep learning with information retrieval to localize buggy files for bug reports (n), Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 476–481.
[23] T.D. LaToza, B.A. Myers, Hard-to-answer questions about code, Evaluation and Usability of Programming Languages and Tools, ACM, 2010, p. 8.
[24] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436.
[25] C. Liu, L. Fei, X. Yan, J. Han, S.P. Midkiff, Statistical debugging: a hypothesis testing-based approach, IEEE Trans. Software Eng. 32 (10) (2006) 831–848.
[26] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. (1947) 50–60.
[27] C.D. Manning, P. Raghavan, H. Schütze, Scoring, term weighting and the vector space model, 100 (2008), pp. 2–4.
[28] M. Mealling, R. Denenberg, Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations, Technical Report, (2002).
[29] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, arXiv:1301.3781 (2013a).
[30] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Advances in Neural Information Processing Systems, (2013), pp. 3111–3119.
[31] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 181–190.
[32] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, AAAI 2 (2016), p. 4.
[33] A.T. Nguyen, T.N. Nguyen, Graph-based statistical language model for code, Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, vol. 1, IEEE, 2015, pp. 858–868.
[34] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, T.N. Nguyen, A topic-based approach for narrowing the search space of buggy files from a bug report, Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on, IEEE, 2011, pp. 263–272.
[35] M. Pagliardini, P. Gupta, M. Jaggi, Unsupervised learning of sentence embeddings using compositional n-gram features, arXiv:1703.02507 (2017).
[36] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, Z. Jin, Building program vector representations for deep learning, International Conference on Knowledge Science, Engineering and Management, Springer, 2015, pp. 547–553.
[37] F. Rahman, P. Devanbu, How, and why, process metrics are better, Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 432–441.
[38] R.K. Saha, M. Lease, S. Khurshid, D.E. Perry, Improving bug localization using structured information retrieval, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, 2013, pp. 345–355.
[39] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, Emf-incquery: an integrated development environment for live model queries, Sci. Comput. Program 98 (2015) 80–99.
[40] L. Van Der Maaten, Accelerating t-sne using tree-based algorithms. J. Mach. Learn. Res. 15 (1) (2014) 3221–3245.
[41] P. Vincent, H. Larochelle, Y. Bengio, P.-A. Manzagol, Extracting and composing robust features with denoising autoencoders, Proceedings of the 25th International Conference on Machine Learning, ACM, 2008, pp. 1096–1103.
[42] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 297–308.
[43] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, Toward deep learning software repositories, Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, IEEE, 2015, pp. 334–345.
[44] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[45] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, H. Mei, Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis, Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 181–190.
[46] Y. Xiao, J. Keung, K.E. Bennin, Q. Mi, Machine translation-based bug localization technique for bridging lexical gap, Inf. Softw. Technol. (2018).
[47] Y. Xiao, J. Keung, Q. Mi, K.E. Bennin, Improving bug localization with an enhanced convolutional neural network, Asia-Pacific Software Engineering Conference (APSEC), 2017 24th, IEEE, 2017, pp. 338–347.
[48] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 689–699.
[49] X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, From word embeddings to document similarities for improved information retrieval in software engineering, Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 404–415.
[50] M.D. Zeiler, Adadelta: an adaptive learning rate method, arXiv:1212.5701 (2012).
[51] S. Zhang, C. Zhang, Software bug localization with Markov logic, Companion Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 424–427.
[52] Y. Zhang, B. Wallace, A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification, arXiv:1510.03820 (2015).
[53] J. Zhou, H. Zhang, D. Lo, Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports, Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 14–24.
[54] Z.-H. Zhou, J. Feng, Deep forest: towards an alternative to deep neural networks, arXiv:1702.08835 (2017).