Bug Report Classification using LSTM architecture for More Accurate Software Defect Locating

Anonymous Author(s)

ABSTRACT

The PI proposes a two-year project to develop an interactive and practical system that locates software defects automatically. This system will alleviate developers' effort in bug finding and improve productivity. In preliminary work, the PI developed a ranking model that ranks all the source code files for a given bug report. A source file at a higher position in the ranked list is more likely to contain defects than a source file at a lower position. The proposed system is an extension of the PI's preliminary work. In this project, the PI propose to 1) introduce a pre-filtering technique to filter out low-quality bug reports; 2) use artificial neural network to measure semantic relationship between bug reports and source code files; 3) implement an interactive 3-D VR user interface on multiple platforms for the system. The first two objectives aim at making the system be more accurate. The third objective aims at making the system be more usable. The PI propose to recruit two student assistants including one graduate and one undergraduate to help develop the system and perform experimental evaluation. The system will be published and will be used in the software engineering (SE) courses at California Sate University San Marcos to help students learn SE concepts in a lively manner.

CCS CONCEPTS

• Computing methodologies → Supervised learning by classification; **• Software and its engineering → Software testing and debugging**; **• Information systems →** *Multilingual and cross-lingual retrieval*;

KEYWORDS

Recurrent neural network, long short-term memory, convolutional neural network, bug localization, bug report

ACM Reference Format:

Anonymous Author(s). 2018. Bug Report Classification using LSTM architecture for More Accurate Software Defect Locating. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

A software *bug report* is a descriptive document used to record the scenario of a software product's unexpected behaviors. It provides information for developers to find the cause, which is usually a coding mistake called *bug* or *defect* [8]. During a software product's life cycle, the development team will usually receive a large number of bug reports. For example, the Eclipse Platform project team received 1,567 bug reports in 2017 alone[1]. On the one hand, bug reports provide developers with helpful information in debugging [9], but on the other, their diversity and uneven qualities can make the bug-fixing process nontrivial [7].

Upon receiving a bug report, the assignee will usually use the report information to reproduce the problem [34] and perform code review [2] to locate the bug. This manual process can be time-consuming [47]. To help developers alleviate such tedious effort, several Information Retrieval (IR)-based automatic approaches have recently been proposed to reduce the bug-search space from the whole source code repository, which may contain thousands of files, to a much smaller range (e.g., a list of several highly recommended files). For example, Lam et al. [33] and Huo et al. [23, 24] use Deep Neural Networks (DNN) to learn to relate source code files to bug reports. Ye et al. [64, 66] develope a learning-to-rank model to combine various *features* for ranking source files. Sahar et al. [55] and Zhou et al. [67] used Vector Space Model (VSM), Kim et al. [30] apply Naïve Bayes, Nguyen et al. [49] and Lukins et al. [40] use Latent Dirichlet Allocation (LDA), Rao et al. [53] apply various IR models including VSM and LDA to measure the relaitonship between bug reports and source files for recommendations.

These IR-based approaches, unlike some other specturm-based approaches [1, 11, 14, 26, 29, 36, 39, 50, 51] 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. [6], in which they receive responses from 446 developers, there is usually a mismatch between what developers consider most helpful and what is provided in the bug reports. The quality of bug report contents can vary remarkably. Bug reports may provide insufficient or even inadequate information for developers to find the cause [6, 20, 30].

Besides, some bug reports can be helpful to developers for manual search but not for IR-based approaches. Take Eclipse bug 305571[2] for example, it reports a problem described as "*Links in forms editors keep getting bolder and bolder*". It provides information to reproduce the problem. Through a serious of intra-group communications, developers reproduced the abnormal scenario, got screenshots, performed manual investigations, and eventually fixed the bug in file *TextHyperlinkSegment.java*. However, this buggy file does not have explicit semantic relationship with the bug report. So when we used the Lucene[3] implementation of VSM to rank all the source files for this report, the buggy file was ranked much

ESEC/FSE 2018, 49 November, 2018, Lake Buena Vista, Florida, United States
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[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

lower than some irrelevant files such as *FormPage.java* and *FormEditor.java* that have greater lexical similarity with the report.

As such, for low-quality reports and reports that do not semantically relate to the bug, instead of running an IR-based ranking system to obtain incorrect recommendations, keeping silent can reduce false positives and increase the average ranking precision.

Kim et al. [30] proposed a two-phase model that first classifies bug reports into either "predictable" or "deficient" and then locates bugs for only "predictable" reports. Their model uses fixed buggy files as labels and applies Naïve Bayes to classify a "predictable" 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 cannot not be located. Despite of this problem, Kim's work inspire us to, before applying a specific IR-based system to find the bug, perform classification to filter out deficient reports and reports that are unhelpful to the IR-based system.

This paper proposes a Long Short-Term Memory (LSTM)-based pre-filtering approach to classify bug reports as either "predictable" or "unpredictable". An LSTM network is a Recurrent Neural Network (RNN) with LSTM units [19] for learning features from sequence data. It has been recently used in the Software Engineering (SE) domain to solve SE problems [10, 23]. We use LSTM to learn from bug reports their vector representations, which are then serve as input *features* to a *Softmax* layer for classification. If a bug report is classified as "predictable", we use an existing IR-based system to help locate the bug for it, otherwise, we keep silent.

We test our classification approach over 11,000 bug reports from four large-scale open source Java projects. Experiments show that, under a trade-off between the classification recall and precision, our approach can help an IR-based bug-locating system achieve better ranking result.

We also perform evaluations to compare LSTM with Convolutional Neural Network (CNN) [37] (another class of DNN that is recently used to solve SE tasks [36, 46, 62]), multilayer perceptron [21], and a simple baseline approach classifying a bug report based on its length with the assumption that larger content may contain more helpful information. Results show that the simple baseline approach can achieve comparably result with multilayer perceptron. LSTM and CNN perform better than the others. LSTM achieves the best trade-off between precision and recall.

The main contributions of this paper include: a bug report pre-filtering model to filter out "unpredictable" reports before running an IR-based system for bug locating; an adaptation of LSTM in the task of bug report classification; extensive evaluations to compare the effectiveness of LSTM with CNN, multilayer perceptron, and a simple baseline approach.

The rest of this paper is structured as follows. Section 2 draws an overall picture of the pre-filtering model for bug locating. Section 3 details the adaptation of an LSTM network for bug report classification. Section 4 introduces CNN, multilayer perceptron, and a simple baseline approach used for comparisons. Section 5 presents the evaluation setup and result. Following a discussion of related work in Section 6, the paper ends in Section 7 with future work and concluding remark.

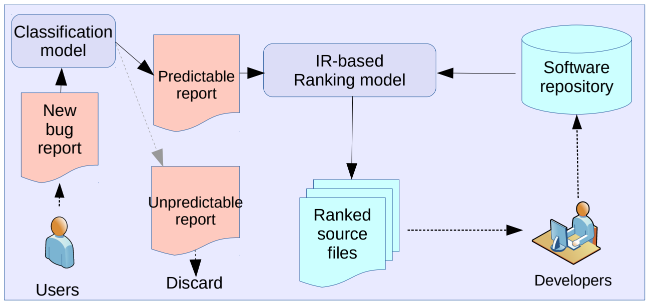

Figure 1: High level architecture: pre-filtering before ranking.

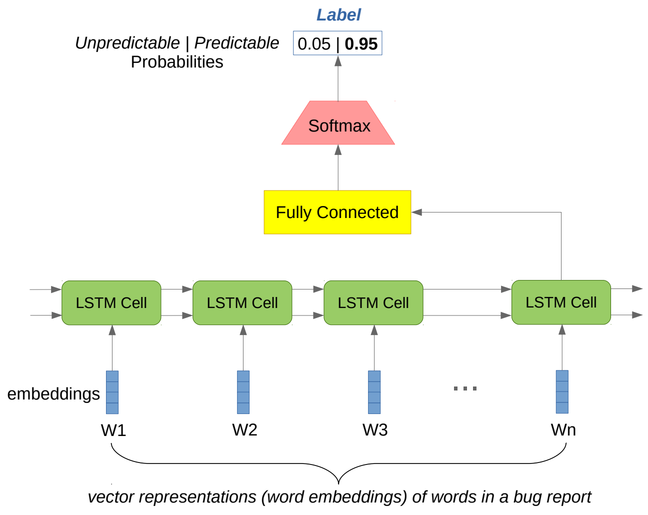
Label
Unpredictable | Predictable Probabilities
0.05 | 0.95
Softmax
Fully Connected
LSTM Cell
LSTM Cell
LSTM Cell
LSTM Cell
embeddings
W1
W2
W3
...
Wn
vector representations (word embeddings) of words in a bug report

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

2 HIGH LEVEL ARCHITECTURE OF BUG REPORT PRE-FILTERING

Figure 1 shows the high level architecture of our bug report pre-filtering approach. When a new bug report is received, it will first be classified by a classification model into one of the two categories: "predictable" and "unpredictable". A "predictable" report is considered as informative and helpful to an IR-based ranking system for bug locating. It serves as input to the ranking system, which uses the report content to rank all the source code files and recommend the top ranked ones as "buggy" to developers to review. An "unpredictable" report, instead, is considered unhelpful to the IR-based ranking system and will be discarded. By keeping silent on unhelpful reports, the ranking system can reduce the number of false positives and make the recommendations be more trustworthy.

3 BUG REPORT CLASSIFICATION USING AN LSTM NETWORK

The architecture of the classification model is shown in Figure 2. Given a bug report, it takes as input the vector representations of words in the report to a Recurrent Neural Network (RNN) implemented with LSTM units. The output of the LSTM unit at the last time step is fed into a fully connected layer, followed by the Softmax model that produces the categorical distribution.

The following subsections detail each step of this process.

3.1 From Bug Report to Bug Report Matrix

Given a bug report, we concatenate its summary and its description into one document. Punctuation and numerical numbers are removed. Then we split the text by white space and obtain a bag-of-words T of the document: $T = (w_1, w_2, w_3, ...w_N)$, where w_i is a word token in the report and N is the total number of words.

Next, we represent every word token w_i with a d-dimensional vector of real numbers $\mathbf{w}_i$ called word embedding that captures some contextual semantic meanings [38]. We use Mikolov's Skip-gram model [35] to learn word embeddings with size of 100 on the Wikipedia data dumps[4]. For unseen words that are not in the Wiki vocabulary, we represent them using a vector that all 100 elements are randomly generated within the range of $(-1, 1)$.

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 set a fixed size of 100 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 100 words will be padded with zero vectors.

Then the original bag-of-words T of a bug report is converted into a matrix of real numbers: $\mathcal{M} \in \mathbb{R}^{100x100}$, where $\mathcal{M} = (\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, ..., \mathbf{w}_{100})$ and $\mathbf{w}_i \in \mathbb{R}^{100}$ 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*) [19]. An LSTM (memory) cell $\mathbf{c}_t \in \mathbb{R}^m$ is a m-dimensional vector that stores m values (states) of the hidden layer at time step t. The three multiplicative gates are used to control the memory of the hidden states and the update of the output. RNNs allow information (weights of connections between the input and the hidden layer) to be accumulated from previous time steps. They are powerful for modeling dependencies in time series [17, 57]. Traditional RNNs are difficulty to train on long sequence due to the vanishing gradient problem [5]. LSTM networks effectively alleviate this problem by using the multiplicative gates to learn long-term dependencies over long periods of time.

The LSTM network takes the bug report matrix $\mathcal{M}$ as a time series input (from $\mathbf{w}_1$ to $\mathbf{w}_{100}$). At each time step, as shown Figure 3, an embedding $\mathbf{w}_i$ is fed into the LSTM network, where the output $\mathbf{h}_i \in \mathbb{R}^m$ of an LSTM unit is determined based on three types of input: the current embedding $\mathbf{w}_i \in \mathbb{R}^{100}$, the previous LSTM output $\mathbf{h}_{i-1} \in \mathbb{R}^m$, and the content of the memory cell $\mathbf{c}_{i-1} \in \mathbb{R}^m$ from the previous time step, where m is the number of hidden units (states) in the memory cell.

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

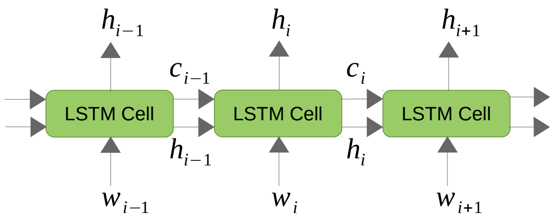

Figure 3: An LSTM Network.

In this paper, the output of the LSTM unit from the last time step $\mathbf{h}_{100} \in \mathbb{R}^m$ is 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 $\mathbf{h} \in \mathbb{R}^m$ is fed into a fully connected layer with rectifier (ReLU) [48] activation function.

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

The output $\mathbf{x} \in \mathbb{R}^n$ of the fully connected layer is shown in Equation 1, where $\mathbf{U} \in \mathbb{R}^{mxn}$ is the weighting matrix initialized using the Glorot uniform scheme [16], $\mathbf{b} \in \mathbb{R}^n$ is the bias, and $n = 2$ is the number of categories. It serves as input to a Softmax model.

Softmax normalizes $\mathbf{x} \in \mathbb{R}^n$ 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: "predictable" and "unpredictable".

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

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 to 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 [31].

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

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

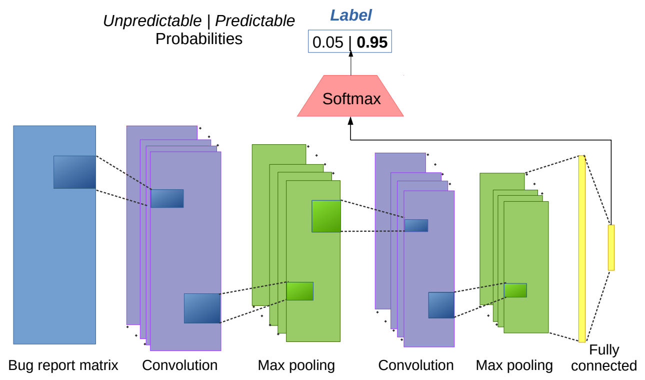

Figure 4: Bug-report classification architecture: using CNN.

training efficiency, helps avoid local minima, and achieves better convergence [17].

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 [15] during training, which reduces overfittings by randomly cleaning up some input, output, and hidden units.

4 BUG REPORT CLASSIFICATION USING CNN, MULTILAYER PERCEPTRON, AND A SIMPLE BASELINE

This section introduces bug report classification using Convolutional Neural Network (CNN), multilayer perceptron, and a simple baseline approach for comparisons with using the LSTM network.

4.1 CNN for Bug Report Classification

A CNN is a deep feedforward neural network composed of one or more convolutional layers with subsamplings (poolings) [37]. 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 powerful for learning long-term dependencies from time series, CNNs learn dependencies from spatial locality and work effectively on 2-D structure (e.g., image) [54].

Figure 4 shows the overall architecture of using CNN for bug report classification. In this paper, we use a CNN with two convolutioan layers with max poolings followed by two fully connected layers with ReLU. It takes as input a bug report matrix $\mathcal{M} \in \mathbb{R}^{100x100}$ 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 minimzing the cross-enropy cost function per mini-batch over a maximum 500 epochs with an early stopping criterion.

4.2 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 [21].

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

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.

4.3 A Simple Baseline Approach for Bug Report Classification

We build a simple baseline approach for bug report classification based on a simple assumption that longer content of a report contains more helpful information related to the bug.

```
def classify(report, threshold):
    get the bag-of-words of the report
    set $n$ = |bag-of-words|
    if $n$ > threshold:
        return "predictable"
    else:
        return "unpredictable"
```

Figure 5: Classifying a bug report based on its length.

The simple baseline approach is shown in Figure 5. If the length of a bug report is greater than a given threshold value, we deem it "predictable", otherwise, "unpredictable".

Unlike the neural-network approaches taking as input a bug report matrix created using word embeddings, this simple baseline approach uses the raw report as input. Although it does not learn any lexical or semantic meanings, it catches a simple but important structural information: the report size. This simple approach is general enough, as a comparison baseline, to work on any types of reports.

5 EVALUATION

5.0.1 C.5 Evaluation. We will evaluate the software defect positioning system on several large-scale open-source projects that contain a sufficient number (more than 2,000) of source code files and previously fixed bug reports. We will conduct experiments on software projects that are written in Java as well as projects written in C/C++ because these programming languages are widely used in both industry and academic. We will use projects from the Eclipse foundation [6] and Apache foundation [7] because 1) both have many large-scale open-source projects written in Java and C/C++; 2) the source code packages can be easily downloaded from their GIT repositories; 3) their bug reports or issue reports are public accessible. More specifically, the projects used in our preliminary work [65] will be used in this study. Additionally, we will run experiments on more projects such as Apache HTTP Server [8] written in C, Lucene [9] (an information retrieval software library) written in Java, and Hadoop [10] (a software framework for distributed storage and bigdata processing) written in Java. The selection of bug reports for evaluation is based on the same heuristics in [12, 64].

We will run the system to rank all the source code files for a given bug report and compare the result with the actual fix. The evaluation metrics such as *Mean Average Precision* (MAP) [41] used in our preliminary work [65] will also be used in this study. Additionally, we will use Normalized Discounted Cumulative Gain (NDCG) [25], which is widely used in evaluating information retrieval models e.g. web search engines, to evaluate our system.

The PI is aware that user study is an effective way to evaluate the effectiveness of the proposed system in developers' real work. However, this is not the main goal of the proposed study. So the PI leave this to future work after the system is published. This study will also provide the basis for our future research on evaluating the effectiveness and usability of the proposed system in assisting teaching software engineering course at CSUSM.

5.1 D. Work Plan

The PI plan to hire one graduate student enrolled in the Master of Science Program in Computer Science (CS) and one undergraduate student enrolled in the Bachelor of Science in Computer Science program at CSUSM to help develop the system. The PI takes the overall responsibility of directing the project and keeps mentoring the students during the development.

In the first year, the graduate student will help implement the bug-report classification model, the LSTM-based semantic similarity feature, and the ranking model running on the server-side. The undergraduate student will implement the client-side programs and the web server that takes charge of the communication between the ranking model and the client-program. Since the ranking model, the database, and the web server work closely, the students will also work together closely during the development.

[6]https://eclipse.org/
[7]https://www.apache.org/
[8]https://httpd.apache.org/
[9]https://lucene.apache.org/
[10]http://hadoop.apache.org/

In the second year, two students will take charge of the maintenance and improvement of the system. Additionally, the graduate student will help run experiments to evaluate the ranking performance of the system on several open-source software projects, analyze the results, and improve the system accordingly. The undergraduate student will help develop a supplemental 3-D VR software visualization module. By the end of this year, the practical system will be published.

Recruitment of students will begin in spring, about three months before they start to work on the project. The PI will broadcast a hiring advertisement on CS-major electronic mailing lists, post a flier on class forums, and make a presentation of the project in the college-wide Frontiers in Science talk. In the coming 2017-2018 academic year, the CS department at CSUSM has a total of 866 undergraduate enrollments and a total of 39 graduate enrollments. Many students have desire to gain hands-on experience by working with faculty members on research projects. The PI will interview interested students. Preference will be given to economically-disadvantaged students, minority students, students making good progress toward their degree, and students who have demonstrated interest in this project.

5.2 E. Broader Impacts

The proposed study will result in a practical software system that alleviates develops' effort in bug finding and improves productivity. The system will be published. Any software developers can use and test it on their own projects. The system will be used in the software engineering courses at CSUSM to help students learn software engineering concepts in a lively manner. This project will expose students to an up-to-date software engineering research topic as well as some cutting-edge techniques including artificial neural networks and virtual reality.

The research outcome of this project will provide the basis for the PI's future research in performing user study and collecting user feedback to evaluate the effectiveness of the system in both academic and industry. The experience learned from this project will be very helpful for PI's research in code recommendation and automatic programming. The methodology and techniques used in this project will contribute to the software engineering research community.

The PI graduated with a Ph.D. degree from Ohio University at May 2016 and joined the CS department at CSUSM as an tenure-track faculty at August 2016. The funding to the proposed study will help the PI obtain important computing resources for building a software engineering (SE) research lab at CSUSM to continue his research. This fund, which supports two student assistants, will also help the PI found an SE research group at CSUSM. The research group will work on adapting new techniques to solve SE tasks, applying up-to-date SE research outcome to assist teaching SE courses, and engaging students in doing SE research projects.

The funding will have a significant impact on the quality of education for students here at CSUSM. The funds requested for student assistants will allow some of our economically-disadvantaged students, may of whom work in retail and on campus dining, to have paid positions during the academic year and summer.

Furthermore, funding of the proposed research program will create new opportunities for our students that have strong desire to participate in research projects. The research opportunities created from the funding of the proposed research will directly contribute to providing talented undergraduates at CSUSM with the research experience necessary to be competitive applicants for top-tier Ph.D. programs.

6 EVALUATION

In this section, we describe an extensive set of experiments that are intended to determine the utility of the new document similarity measures based on word embeddings in the context of bug localization. This is an information retrieval task in which queries are bug reports and the system is trained to identify relevant, buggy files.

6.1 Text Pre-processing

There are three types of text documents used in the experimental evaluations in this section: 1) the Eclipse API reference, developer guides, Java API reference, and Java tutorials that are used to train the word embeddings; 2) the bug reports; and 3) the source code files. When creating bag-of-words for these documents, we use the same pre-processing steps on all of them: we remove punctuation and numerical numbers, then split the text by whitespace.

The tokenization however is done differently for each category of documents. In the one-vocabulary setting, compound words in the bug reports and the source code files are split based on capital letters. For example, "WorkbenchWindow" is split into "Workbench" and "Window", while its original form is also reserved. We then apply the Porter stemmer on all words/tokens.

In the two-vocabulary setting, a code token such as a method name "*clear* is marked as "@clear@" so that it can be distinguished from the adjective "clear". Then we stem only the natural language words. We also split compound natural language words. In order to separate code tokens from natural language words in the training corpus, we wrote a dedicated HTML parser to recognize and mark the code tokens. For bug reports, we mark words that are not in an English dictionary as code tokens. For source code files, all tokens except those in the comments are marked as code tokens. Inside the comments, words that are not in an English dictionary are also marked as code tokens.

6.2 Corpus for Training Word Embeddings

To train the shared embeddings, we created a corpus from documents in the following Eclipse repositories: the Platform API Reference, the JDT API Reference, the Birt API Reference, the Java SE 7 API Reference, the Java tutorials, the Platform Plug-in Developer Guide, the Workbench User Guide, the Plug-in Development Environment Guide, and the JDT Plug-in Developer Guide. The number of documents and words/tokens in each repository are shown in Table 2. All documents are downloaded from their official website[11][12]. Code tokens in these documents are usually placed between special HTML tags such as ⟨*code*⟩ or emphasized with different fonts.

[11]http://docs.oracle.com/javase/7/docs
[12]http://www.eclipse.org/documentation

Table 2: Documents for training word embeddings.

Data sources	Documents	Words/Tokens
Platform API Reference	3,731	1,406,768
JDT API Reference	785	390,013
Birt API Reference	1,428	405,910
Java SE 7 API Reference	4,024	2,840,492
The Java Tutorials	1,282	1,024,358
Platform Plug-in Developer Guide	343	182,831
Workbench User Guide	426	120,734
Plug-in Development Environment Guide	269	90,356
JDT Plug-in Developer Guide	164	64,980
Total	12,452	6,526,442

Table 3: The vocabulary size.

Word embeddings trained on:	Vocabulary size
one-vocabulary setting	21,848
two-vocabulary setting	25,676

Table 4: Number of word pairs.

Approach	# of word pairs
One-vocabulary embeddings	238,612,932
Two-vocabulary embeddings	329,615,650
SEWordSim [59]	5,636,534
SWordNet [63]	1,382,246

To learn the shared embeddings, we used the Skip-gram model, modified such that it works in the training scenarios described in Section **??**. Table 3 shows the number of words in each vocabulary setting. Table 4 compares the number of word pairs used to train word embeddings in the one- and two-vocabulary settings with the number of word pairs used in two related approaches. Thus, when word embeddings are trained on the one-vocabulary setting, the vocabulary size is 21,848, which leads to 238,612,932 word pairs during training. This number is over 40 times the number of word pairs in SEWordSim [59], and is more than 172 times the number of word pairs in SWordNet [63].

6.3 Benchmark Datasets

We perform evaluations on the fined-grained benchmark dataset from [64]. Specifically, we use four open-source Java projects: Birt[13], Eclipse Platform UI[14], JDT[15], and SWT[16]. For each of the 10,000 bug reports in this dataset, we *checkout* a before-fixed version of the source code, within which we rank all the source code files for the specific bug report.

Since the training corpus for word embeddings (shown in Table 2) contains only Java SE 7 documents, for testing we use only bug reports that were created for Eclipse versions starting with 3.8, which is when Eclipse started to add Java SE 7 support. The Birt, JDT, and SWT projects are all Eclipse Foundation projects, and also support Java SE 7 after the Eclipse 3.8 release. Overall, we collect

[13]https://www.eclipse.org/birt/
[14]http://projects.eclipse.org/projects/eclipse.platform.ui
[15]http://www.eclipse.org/jdt/
[16]http://www.eclipse.org/swt/

Table 1: Benchmark Projects: *Eclipse* refers to Eclipse Platform UI.*

Project	Time Range	# of bug reports used for testing	# of bug reports used for training	# of bug reports used for tuning	total
Birt	2005-06-14 – 2013-12-19	583	500	1,500	2,583
Eclipse*	2001-10-10 – 2014-01-17	1,656	500	1,500	3,656
JDT	2001-10-10 – 2014-01-14	632	500	1,500	2,632
SWT	2002-02-19 – 2014-01-17	817	500	1,500	2,817

for testing 583, 1656, 632, and 817 bug reports from Birt, Eclipse Platform UI, JDT, and SWT, respectively. Older bug reports that were reported for versions before release 3.8 are used for training and tuning the learning-to-rank systems.

Table 1 shows the number of bug reports from each project used in the evaluation. The methodology used to collect the bug reports is discussed at length in [64]. Here we split the bug reports into a testing set, a training set, and a tuning set. Taking Eclipse Platform UI for example, the newest 1,656 bug reports, which were reported starting with Eclipse 3.8, are used for testing. The older 500 bug reports in the training set are used for learning the weight parameters of the ranking function in Equation **??**, using the SVM^{rank} package [27, 28]. The oldest 1,500 bug reports are used for tuning the hyper-parameters of the Skip-gram model and the SVM^{rank} model, by repeatedly training on 500 and testing on 1000 bug reports. To summarize, we tune the hyper-parameters of the Skip-gram model and the SVM^{rank} model on the tuning dataset, then train the weight vector used in the ranking function on the training dataset, and finally test and report the ranking performance on the testing dataset. After tuning, the Skip-gram model was train to learn embeddings of size 100, with a context window of size 10, a minimal word count of 5, and a negative sampling of 25 words.

6.4 Results and Analysis

We ran extensive experiments for the bug localization task, in order to answer the following research questions:

RQ1: Do word embeddings help improve the ranking performance, when added to an existing strong baseline?

RQ2: Do word embeddings trained on different corpora change the ranking performance?

RQ3: Do the word embedding training heuristics improve the ranking performance, when added to the vanilla Skip-gram model?

RQ4: Do the modified text similarity functions improve the ranking performance, when compared with the original similarity function in [44]?

We use the Mean Average Precision (MAP) [41], which is the mean of the average precision values for all queries, and the Mean Reciprocal Rank (MRR) [60], which is the harmonic mean of ranks of the first relevant documents, as the evaluation metrics. MAP and MRR are standard evaluation metrics in IR, and were used previously in related work on bug localization [55, 64, 67].

6.4.1 ***RQ1:*** *Do word embeddings help improve the ranking performance?* The results shown in Table 5 compare the **LR** system introduced in [64] with a number of systems that use word embeddings in the one- and two-vocabulary settings, as follows: **LR+WE**^{1} refers to combining the one-vocabulary word-embedding-based features with the six features of the LR system from [64], **LR+WE**^{2} refers to combining the two-vocabulary word-embedding-based features with the LR system, **WE**^{1} refers to using only the one-vocabulary word-embedding-based features, and **WE**^{2} refers to using only the two-vocabulary word-embedding-based features. The parameter vector of each ranking system is learned automatically. The results show that the new word-embedding-based similarity features, when used as additional features, improve the performance of the **LR** system. The results of both **LR+WE**^{1} and **LR+WE**^{2} show that the new features help achieve 8.1%, 20%, 5.6%, and 16.7% relative improvements in terms of MAP over the original **LR** approach, for Eclipse Platform UI, JDT, SWT, and Birt respectively. In [64], **LR** was reported to outperform other state-of-the-art bug localization models such as the VSM-based BugLocator from Zhou et al. [67] and the LDA-based BugScout from Nguyen et al. [49].

Table 5: MAP and MRR for the 5 ranking systems.

Project	Metric	LR+WE^1 ϕ_1-ϕ_8	LR+WE^2 ϕ_1-ϕ_8	LR ϕ_1-ϕ_6	WE^1 ϕ_7-ϕ_8	WE^2 ϕ_7-ϕ_8
Eclipse Platform UI	MAP	0.40	0.40	0.37	0.26	0.26
	MRR	0.46	0.46	0.44	0.31	0.31
JDT	MAP	0.42	0.42	0.35	0.22	0.23
	MRR	0.51	0.52	0.43	0.27	0.29
SWT	MAP	0.38	0.38	0.36	0.25	0.25
	MRR	0.45	0.45	0.43	0.30	0.30
Birt	MAP	0.21	0.21	0.19	0.13	0.13
	MRR	0.27	0.27	0.24	0.17	0.17

Another observation is that using word embeddings trained on one-vocabulary and using word embeddings trained on two-vocabulary achieve almost the same results. By looking at a sample of API documents and code, we discovered that class names, method names, and variable names are used with a consistent meaning throughout. For example, developers use *Window* to name a class that is used to create a window instance, and use *open* to name a method that performs an open action. Therefore, we believe the two-vocabulary setting will be more useful when word embeddings are trained on both software engineering (SE) and natural language (NL) corpora (e.g. Wikipedia), especially in situations in which a word has NL meanings that do not align well with its SE meanings. For example, since *eclipse* is used in NL mostly with the astronomical sense, it makes sense for *eclipse* to be semantically more similar with *light* than *ide*. However, in SE, we want *eclipse* to be more similar to *ide* and *platform* than to *total*, *color*, or *light*. By training separate embeddings for *eclipse* in NL contexts (i.e. *eclipse_NL*) vs. *eclipse* in SE contexts (i.e. *eclipse_SE*), the expectation is that, in an SE setting,

Table 6: Results on easy (T1) vs. difficult (T2) bug reports, together with # of bug reports (size) and average # of relevant files per bug report (avg).

		T1		T2	
		LR+WE^1	LR	LR+WE^1	LR
Eclipse	Size/Avg	322/2.11		1,334/2.89	
	MAP	0.80	0.78	0.30	0.27
	MRR	0.89	0.87	0.36	0.33
JDT	Size/Avg	84/2.60		548/2.74	
	MAP	0.79	0.75	0.36	0.29
	MRR	0.90	0.87	0.45	0.37
SWT	Size/Avg	376/2.35		441/2.57	
	MAP	0.57	0.55	0.22	0.21
	MRR	0.66	0.65	0.27	0.26
Birt	Size/Avg	27/2.48		556/2.24	
	MAP	0.48	0.54	0.20	0.17
	MRR	0.62	0.69	0.25	0.22

the *eclipse_SE* embedding would be more similar with the *ide_SE* embedding than the *total_SE* or *color_SE* embeddings.

Kochhar et al. [32] reported from an empirical study that the localized bug reports, which explicitly mention the relevant file names, "*statistically significantly and substantially*" impact the bug localization results. They suggested that there is no need to run automatic bug localization techniques on these bug reports. Therefore, we separate the testing bug reports for each project into two subsets T1 (easy) and T2 (difficult). Bug reports in T1 mention either the relevant file names or their top-level public class names, whereas T2 contains the other bug reports. Note that, although bug reports in T1 make it easy for the programmer to find a relevant buggy file, there may be other relevant files associated with the same bug report that could be more difficult to identify, as shown in the statistics from Table 6.

Table 6 shows the MAP and MRR results on T1 and T2. Because **LR+WE**^{1} and **LR+WE**^{2} are comparable on the test bug reports, here we compare only **LR+WE**^{1} with **LR**. The results show that both **LR+WE**^{1} and **LR** achieve much better performance on bug reports in T1 than T2 for all projects. This confirms the conclusions of the empirical study from Kochhar et al. [32]. The results in Table 6 also show that overall using word embeddings helps on both T1 and T2. One exception is Birt, where the use of word embeddings hurts performance on the 27 easy bugs in T1, a result that deserves further analysis in future work.

To summarize, we showed that using word embeddings to create additional semantic similarity features helps improve the ranking performance of a state-of-the-art approach to bug localization. However, separating the code tokens from the natural language words in two vocabularies when training word embeddings on the SE corpus did not improve the performance. In future work, we plan to investigate the utility of the two-vocabulary setting when training with both SE and NL corpora.

6.4.2 ***RQ2:*** *Do word embeddings trained on a different corpora change the ranking performance?* To test the impact of the training corpus, we train word embeddings in the one-vocabulary setting using the Wiki data dumps[17], and redo the ranking experiment.

Table 7: The size of the different corpora.

Corpus	Vocabulary	Words/Tokens
Eclipse and Java	21,848	6,526,442
Wiki	2,098,556	3,581,771,341

Table 8: Comparison of the LR+WE^1 results when using word embeddings trained on different corpora.

Corpus	Metric Metric	Eclipse Platform UI	JDT	SWT	Birt
Eclipse and Java documents	MAP	0.40	0.42	0.38	0.21
	MRR	0.46	0.51	0.45	0.27
Wiki	MAP	0.40	0.41	0.38	0.21
	MRR	0.46	0.51	0.45	0.27

The advantage of using the Wiki corpus is its large size for training. Table 7 shows the size of the Wiki corpus. The number of words/tokens in the Wiki corpus is 548 times the number in our corpus, while its vocabulary size is 96 times the vocabulary size of our corpus. Theoretically, the larger the size of the training corpus the better the word embeddings. On the other hand, the advantage of the smaller training corpus in Table 2 is that its vocabulary is close to the vocabulary used in the queries (bug reports) and the documents (source code files).

Table 8 shows the ranking performance by using the Wiki embeddings. Results show that the project specific embeddings achieve almost the same MAP and MRR for all projects as the Wiki embeddings. We believe one reason for the good performance of the Wiki embeddings is the pre-processing decision to split compound words such as `WorkbenchWindow` that do not appear in the Wiki vocabulary into their components words `Workbench` and `Window`, which belong to the Wiki vocabulary. Correspondingly, Table 9 below shows the results of evaluating just the word-embeddings features (**WE**^{1}) on the Eclipse project with the two types of embeddings, with and without splitting compound words. As expected, the project-specific embeddings have better performance than the Wiki-trained embeddings when compound words are not split; the comparison is reversed when splitting is used. Overall, each cor-

Table 9: Project-specific vs. Wikipedia embeddings performance of WE^1 features, with and without splitting compound words.

Project	Metric	No Split	Split
Eclipse/Java	MAP	0.254	0.260
	MRR	0.307	0.310
Wikipedia	MAP	0.248	0.288
	MRR	0.300	0.346

[17]https://dumps.wikimedia.org/enwiki/

Table 10: LR+WE[1] results obtained using the enhanced vs. the original Skip-gram model.

Project	Metric	LR ϕ_1-ϕ_8	Enhanced Skip-gram ϕ_1-ϕ_6	Original Skip-gram ϕ_1-ϕ_8
Eclipse Platform UI	MAP	0.37	0.40	0.40
	MRR	0.44	0.46	0.46
JDT	MAP	0.35	0.42	0.42
	MRR	0.43	0.51	0.51
SWT	MAP	0.36	0.38	0.37
	MRR	0.43	0.45	0.44
Birt	MAP	0.19	0.21	0.21
	MRR	0.24	0.27	0.27

pus has its own advantages: while the embeddings trained on the project-specific corpus may better capture specific SE meanings, the embeddings trained on Wikipedia may benefit from the substantially larger amount of training examples. Given the complementary advantages, in future work we plan to investigate training strategies that exploit both types of corpora.

6.4.3 ***RQ3:*** *Do the word embedding training heuristics improve the ranking performance?* Table 10 shows the results of using the original Skip-gram model without applying the heuristic techniques discussed in Sections **??** and **??**. It shows that both the enhanced and the original Skip-gram model achieve the same results most of the time. These results appear to indicate that increasing the number of training pairs for word embeddings will not lead to further improvements in ranking performance, which is compatible with the results of using the Wiki corpus vs. the much smaller project-specific corpora.

6.4.4 ***RQ4:*** *Do the modified text similarity functions improve the ranking performance?* Table 11 below compares the new text similarity functions shown in Equation **??** with the original text similarity function from Mihalcea et al. [44], shown in Equation **??**. In $\mathbf{WE}^1_{ori}$, the new features ϕ_7 and ϕ_8 are calculated using the one-vocabulary word embeddings and the original idf-weighted text similarity function. The results of **LR+WE**[1] and **LR** are copied from Table 5, for which ϕ_7 and ϕ_8 are calculated using the new text similarity functions.

Table 11: Comparison between the new text similarity function (LR+WE[1]) and the original similarity function (LR+$\mathbf{WE}^1_{ori}$).

Project	Metric	LR ϕ_1-ϕ_8	LR+WE[1] ϕ_1-ϕ_6	LR+WE^1_{ori} ϕ_7-ϕ_8
Eclipse Platform UI	MAP	0.37	0.40	0.37
	MRR	0.44	0.46	0.43
JDT	MAP	0.35	0.42	0.36
	MRR	0.43	0.51	0.45
SWT	MAP	0.36	0.38	0.37
	MRR	0.43	0.45	0.44
Birt	MAP	0.19	0.21	0.20
	MRR	0.24	0.27	0.25

Results show that the new text similarity features lead to better performance than using the original text similarity function. The new features obtain a 20% relative improvement in terms of MAP over the **LR** approach, while features calculated based on the original text similarity function achieve only a 3% relative improvement.

7 EVALUATION OF WORD EMBEDDINGS FOR API RECOMMENDATION

To assess the generality of using document similarities based on word embeddings for information retrieval in software engineering, we evaluate the new similarity functions on the problem of linking API documents to Java questions posted on the community question answering (cQA) website Stack Overflow (SO). The SO website enables users to ask and answer computer programming questions, and also to vote on the quality of questions and answers posted on the website. In the Question-to-API (Q2API) linking task, the aim is to build a system that takes as input a user's question in order to identify API documents that have a non-trivial semantic overlap with the (as yet unknown) correct answer. We see such a system as being especially useful when users ask new questions, for which they would have to wait until other users post their answers. Recommending relevant API documents to the user may help the user find the answer on their own, possibly even before the correct answer is posted on the website. To the best of our knowledge, the Q2API task for cQA websites has not been addressed before.

In order to create a benchmark dataset, we first extracted all questions that were tagged with the keyword 'java', using the datadump archive available on the Stack Exchange website. Of the 1,493,883 extracted questions, we used a script to automatically select only the questions satisfying the following criteria:

(1) The question score is larger than 20, which means that more than 20 people have voted this question as "useful".
(2) The question has answers of which one was checked as the "correct" answer by the user who asked the question.
(3) The "correct" answer has a score that is larger than 10, which means that more than 10 people gave a positive vote to this answer.
(4) The "correct" answer contains at least one link to an API document in the official Java SE API online reference (versions 6 or 7).

This resulted in a set of high quality 604 questions, whose correct answers contain links to Java API documents. We randomly selected 150 questions and asked two proficient Java programmers to label the corresponding API links as *helpful* or *not helpful*. The remaining 454 questions were used as a (noisy) training dataset. Out of the 150 randomly sampled questions, the 111 questions that were labeled by both annotators as having *helpful* API links were used for testing. The two annotators were allowed to look at the correct answer in order to determine the semantic overlap with the API document.

Although we allow API links to both versions 6 and 7, we train the word embeddings in the one-vocabulary setting, using only the Java SE 7 API documentations and tutorials. There are 5,306 documents in total, containing 3,864,850 word tokens.

We use the Vector Space Model (VSM) as the baseline ranking system. Given a question T, for each API document S we calculate the VSM similarity as feature $\phi_1(T, S)$ and the asymmetric semantic similarities that are based on word embeddings as features $\phi_2(T, S)$ and $\phi_3(T, S)$. In the VSM+WE system, the file score of each API document is calculated as the weighted sum of these three features, as shown in Equation **??**. During training on the 454 questions, the objective of the learning-to-rank system is to find weights such that, for each training question, the relevant (helpful) API documents are ranked at the top. During evaluation on the 111 questions in the test dataset, we rank all the Java API documents S for each question T in descending order of their ranking score $f(T, S)$.

Table 12: Results on the Q2API task.

Approach	MAP	MRR
VSM	0.11	0.12
VSM+WE	0.35	0.39

Table 12 shows the MAP and MRR performance of the baseline VSM system that uses only the VSM similarity feature, vs. the performance of the VSM+WE system that also uses the two semantic similarity features. The results in this table indicate that the document similarity features based on word embeddings lead to substantial improvements in performance. As such, these results can serve as an additional empirical validation of the utility of word embeddings for information retrieval tasks in software engineering.

We note that these results are by no means the best results that we expect for this task, especially since the new features were added to a rather simple VSM baseline. For example, instead of treating SO questions only as bags of undifferentiated words, the questions could additionally be parsed in order to identify code tokens or code-like words that are then disambiguated and mapped to the corresponding API entities [56**?** **?**]. Given that, like VSM, these techniques are highly lexicalized, we expect their performance to improve if used in combination with additional features based on word embeddings.

8 RELATED WORK

Related work on word embedding in NLP was discussed in Section **??**. In this section we discuss other methods for computing word similarities in software engineering and related approaches for bridging the lexical gap in software engineering tasks.

8.1 Word Similarities in SE

To the best of our knowledge, word embedding techniques have not been applied before to solve information retrieval tasks in SE. However, researchers [22, 61, 63] have proposed methods to infer semantically related software terms, and have built software-specific word similarity databases [58, 59].

Tian et al. [58, 59] introduce a software-specific word similarity database called SEWordSim that was trained on StackOverflow questions and answers. They represent words in a high-dimensional space in which every element within the vector representation of word w_i is the Positive Pointwise Mutual Information (PPMI) between w_i and another word w_j in the vocabulary. Because the vector space dimension equals the vocabulary size, the scalability of their vector representation is limited by the size of the vocabulary. When the size of the training corpus grows, the growing vector dimension will lead to both larger time and space complexities. Recent studies [4, 45] also showed that this kind of traditional count-based language models were outperformed by the neural-network-based low-dimensional word embedding models on a wide range of word similarity tasks.

Howard et al. [22] and Yang et al. [63] infer semantically related words directly from comment-code, comment-comment, or code-code pairs without creating the distributional vector representations. They first need to map a line of comment (or code) to another line of comment (or code), and then infer word pairs from these line pairs. Similarly, Wang et al. [61] infer word similarities from tags in FreeCode. The main drawback of these approaches is that they rely solely on code, comments, and tags. More general free-text contents are ignored. Many semantically related words (e.g. "placeholder" and "view") are not in the source code but in the free-text contents of project documents (e.g. the Eclipse user guide, developer guide, and API document shown in Figure **??** to Figure **??**). However, these types of documents are not exploited in these approaches.

More importantly, all the above approaches did not explain how word similarities can be used to estimate document similarities. They reported user studies in which human subjects were recruited to evaluate whether the word similarities are accurate. However, these subjective evaluations do not tell whether and how word similarities can be used in solving IR tasks in SE.

8.2 Bridging the Lexical Gap to Support Software Engineering Tasks

Text retrieval techniques have been shown to help in various SE tasks [18, 42]. However, the system performance is usually suboptimal due to the lexical gap between user queries and code [43]. To bridge the lexical gap, a number of approaches [3, 13, 43, 64**?** **?**] have been recently proposed that exploit information from API documentations. These approaches extract API entities referenced in code, and use the corresponding documentations to enhance the ranking results.

Specifically, McMillan et al. [43] measure the lexical similarity between the user query and API entities, then rank higher the code that uses the API entities with higher similarity scores. Bajracharya et al. [3] augment the code with tokens from other code segments that use the same API entries. Ye et al. [64] concatenate the descriptions of all API entries used in the code, and directly measure the lexical similarity between the query and the concatenated document. The main drawback of these approaches is that they consider only the API entities used in the code. The documentations of other API entities are not used. Figure **??** shows the Eclipse bug 384108. Figure **??** shows its relevant file *PartServiceImpl.java*. Figure **??** shows the description of an API entry *IPageLayout*. Although *IPageLayout* is not used in *PartServiceImpl.java*, its API descriptions contains useful information that can help bridge the lexical gap by mapping the term "view" in bug 384108 with the term "placeholder" in *PartServiceImpl.java*. Therefore, to bridge the lexical gap, we should consider not only the descriptions of the API entities used in the code but also all API documents and project documents (e.g. the user guide shown in Figure **??** and the developer guide in Figure **??**) that are available.

Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA) have been used in the area of feature location and bug localization. Poshyvanyk et al. [51, 52] use LSI to reduce the dimension of the term-document matrix, represent code and queries as vectors, and estimate the similarity between code and queries using the cosine similarity between their vector representations. Similarly, Nguyen et al. [49] and Lukins et al. [40] use LDA to represent code and queries as topic distribution vectors. Rao et al. [53] compare various IR techniques on bug localization, and report that traditional IR techniques such as VSM and Unigram Model (UM) outperform the more sophisticated LSI and LDA techniques. These approaches create vector representations for documents instead of words and estimate query-code similarity based on the cosine similarity between their vectors. McMillan et al. [50] introduced a LSI-based approach for measuring program similarity, and showed that their model achieve higher precision than a LSA-based approach in detecting similar applications. All these works neither measure word similarities nor try to bridge the lexical gap between code and queries.

9 FUTURE WORK

This is the future-work section.

10 CONCLUSION

This is the conclusion section.

REFERENCES

[1] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA, 712–721. http://dl.acm.org/citation.cfm?id=2486788.2486882

[3] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. New York, NY, USA, 157–166.

[4] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Baltimore, Maryland, 238–247. http://www.aclweb.org/anthology/P14-1023

[5] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning Long-term Dependencies with Gradient Descent is Difficult. *Trans. Neur. Netw.* 5, 2 (March 1994), 157–166. https://doi.org/10.1109/72.279181

[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. New York, NY, USA, 308–318.

[7] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work (CSCW '10)*. New York, NY, USA, 301–310.

[8] Bernd Bruegge and Allen H. Dutoit. 2009. *Object-Oriented Software Engineering Using UML, Patterns, and Java* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[9] Raymond P. L. Buse and Thomas Zimmermann. 2012. Information Needs for Software Development Analytics. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. Piscataway, NJ, USA, 987–996.

[10] M. Choetkiertikul, H. K. Dam, T. Tran, T. T. M. Pham, A. Ghose, and T. Menzies. 2018. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering* PP, 99 (2018), 1–1. https://doi.org/10.1109/TSE.2018.2792473

[11] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. New York, NY, USA, 342–351.

[12] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. New York, NY, USA, 433–436.

[13] Tathagata Dasgupta, Mark Grechanik, Evan Moritz, Bogdan Dit, and Denys Poshyvanyk. 2013. Enhancing Software Traceability by Automatically Expanding Corpora with Relevant Documentation. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. Washington, DC, USA, 320–329.

[14] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. 2013. Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software. *Empirical Softw. Engg.* 18, 2 (April 2013), 277–309.

[15] Yarin Gal and Zoubin Ghahramani. 2016. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 1027–1035. http://dl.acm.org/citation.cfm?id=3157096.3157211

[16] Xavier Glorot and Yoshua Bengio. 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*. 249–256. http://www.jmlr.org/proceedings/papers/v9/glorot10a.html

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

[18] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 842–851. http://dl.acm.org/citation.cfm?id=2486788.2486898

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[20] Pieter Hooimeijer and Westley Weimer. 2007. Modeling Bug Report Quality. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. New York, NY, USA, 34–43.

[21] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 2, 5 (July 1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[22] Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. 2013. Automatically Mining Software-based, Semantically-similar Words from Comment-code Mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 377–386. http://dl.acm.org/citation.cfm?id=2487085.2487155

[23] Xuan Huo and Ming Li. 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 (IJCAI'17)*. AAAI Press, 1909–1915. http://dl.acm.org/citation.cfm?id=3172077.3172153

[24] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 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 (IJCAI'16)*. AAAI Press, 1606–1612. http://dl.acm.org/citation.cfm?id=3060832.3060845

[25] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated Gain-based Evaluation of IR Techniques. *ACM Trans. Inf. Syst.* 20, 4 (Oct. 2002), 422–446. https://doi.org/10.1145/582415.582418

[26] Wei Jin and Alessandro Orso. 2013. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. New York, NY, USA, 213–223.

[27] Thorsten Joachims. 2002. Optimizing Search Engines Using Clickthrough Data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*. New York, NY, USA, 133–142.

[28] Thorsten Joachims. 2006. Training Linear SVMs in Linear Time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. New York, NY, USA, 217–226.

[29] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. New York, NY, USA, 273–282.

[30] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Trans. Softw. Eng.* 39, 11 (Nov. 2013), 1597–1610.

[31] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[32] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential Biases in Bug Localization: Do They Matter?. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 803–814. https://doi.org/10.1145/2642937.2642997

[33] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 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)*. 476–481. https://doi.org/10.1109/ASE.2015.73

[34] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer Questions About Code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU '10)*. New York, NY, USA, Article 8, 6 pages.

[35] Quoc V. Le and Tomas Mikolov. 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*. 1188–1196. http://jmlr.org/proceedings/papers/v32/le14.html

[36] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[37] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[38] Omer Levy and Yoav Goldberg. 2014. Neural Word Embedding as Implicit Matrix Factorization. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger (Eds.). Curran Associates, Inc., 2177–2185. http://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf

[39] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 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 (ESEC/FSE-13)*. New York, NY, USA, 286–295.

[40] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2010. Bug Localization Using Latent Dirichlet Allocation. *Inf. Softw. Technol.* 52, 9 (Sept. 2010), 972–990.

[41] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[42] A Marcus and G Antoniol. 2012. On the use of text retrieval techniques in software engineering. In *Proceedings of 34th IEEE/ACM International Conference on Software Engineering, Technical Briefing*.

[43] C. McMillan, M. Grechanik, D. Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *Software Engineering, IEEE Transactions on* 38, 5 (Sept 2012), 1069–1087. https://doi.org/10.1109/TSE.2011.84

[44] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. 2006. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st national conference on Artificial intelligence (AAAI'06)*. AAAI Press, 775–780.

[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proc. of Workshop at ICLR '13*.

[46] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293. http://dl.acm.org/citation.cfm?id=3015812.3016002

[47] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2013. The Design of Bug Fixes. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA, 332–341. http://dl.acm.org/citation.cfm?id=2486788.2486833

[48] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, USA, 807–814. http://dl.acm.org/citation.cfm?id=3104322.3104425

[49] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 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 (ASE '11)*. Washington, DC, USA, 263–272. https://doi.org/10.1109/ASE.2011.6100062

[50] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. 2013. Concept Location Using Formal Concept Analysis and Information Retrieval. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 23 (Feb. 2013), 34 pages.

[51] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. Softw. Eng.* 33, 6 (June 2007), 420–432.

[52] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2006. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*. Washington, DC, USA, 137–148.

[53] Shivani Rao and Avinash Kak. 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 (MSR '11)*. New York, NY, USA, 43–52.

[54] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. 2014. ImageNet Large Scale Visual Recognition Challenge. *ArXiv e-prints* (Sept. 2014). arXiv:cs.CV/1409.0575

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

[56] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 643–652. https://doi.org/10.1145/2568225.2568313

[57] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173

[58] Yuan Tian, D. Lo, and J. Lawall. 2014. Automated construction of a software-specific word similarity database. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*. 44–53. https://doi.org/10.1109/CSMR-WCRE.2014.6747213

[59] Yuan Tian, David Lo, and Julia Lawall. 2014. SEWordSim: Software-specific Word Similarity Database. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 568–571. https://doi.org/10.1145/2591062.2591071

[60] Ellen M. Voorhees. 1999. The TREC-8 Question Answering Track Report. In *In Proceedings of TREC-8*. 77–82.

[61] Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. Inferring Semantically Related Software Terms and Their Taxonomy by Leveraging Collaborative Tagging. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM) (ICSM '12)*. IEEE Computer Society, Washington, DC, USA, 604–607. https://doi.org/10.1109/ICSM.2012.6405332

[62] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 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 (ASE 2016)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2970276.2970357

[63] Jinqiu Yang and Lin Tan. 2012. Inferring semantically related words from software context. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. 161–170. https://doi.org/10.1109/MSR.2012.6224276

[64] Xin Ye, Razvan Bunescu, and Chang Liu. 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 (FSE 2014)*. New York, NY, USA, 689–699. http://dl.acm.org/citation.cfm?id=2337223.2337226

[65] Xin Ye, Razvan Bunescu, and Chang Liu. 2016. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. *IEEE Transactions on Software Engineering* 42, 4 (April 2016), 379–402. https://doi.org/10.1109/TSE.2015.2479232

[66] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/2884781.2884862

[67] Jian Zhou, Hongyu Zhang, and David Lo. 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 (ICSE 2012)*. Piscataway, NJ, USA, 14–24. http://dl.acm.org/citation.cfm?id=2337223.2337226