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

Anonymous Author(s)

ABSTRACT

Recently many information retrieval (IR)-based approaches have been proposed to help locate software defects automatically by using information from bug report contents. However, some bug reports that do not semantically related to the relevant code are not helpful to IR-based systems. Runing an IR-based system on these reports may produce false positives. In this paper, we propose a classification model for classifying a bug report as either helpful or unhelpful using a LSTM-network. By filtering our unhelpful reports before runing an IR-based bug locating system, our approach helps reduce false positives and improve the ranking performance. We test our model over 9,000 bug reports from three software projects. The evaluation result shows that our model helps improve a state-of-the-art IR-based system's ranking performance under a trade-off between the precision and the recall. Our comparison experiments show that the LSTM-network achieves the best trade-off between precision and recall than other classification models including CNN, multilayer perceptron, and a simple baseline approach that classifies a bug report based its length. In the situation that precision is more important than recall, our classification model helps for bug locating.

CCS CONCEPTS

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

KEYWORDS

Long short-term memory, convolutional neural network, bug locating, 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, 11 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* [9]. 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 [10], but on the other, their diversity and uneven qualities can make the bug-fixing process nontrivial [8].

Upon receiving a bug report, the assignee will usually use the report information to reproduce the problem [33] and perform code review [4] to locate the bug. This manual process can be time-consuming [43]. 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. [30] and Huo et al. [24, 25] use Deep Neural Networks (DNN) to learn to relate source code files to bug reports. Ye et al. [57, 58] develope a learning-to-rank model to combine various *features* for ranking source files. Sahar et al. [51] and Zhou et al. [61] used Vector Space Model (VSM), Kim et al. [28] apply Naïve Bayes, Nguyen et al. [45] and Lukins et al. [39] use Latent Dirichlet Allocation (LDA), Rao et al. [49] 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 [3, 12, 14, 26, 27, 35, 38, 47, 48] 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, 21, 28].

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. [28] 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 [20] for learning features from sequence data. It has been recently used in the Software Engineering (SE) domain to solve SE problems [11, 24]. 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 three 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) [36] (another class of DNN that is recently used to solve SE tasks [35, 42, 55]), multilayer perceptron [22], 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.

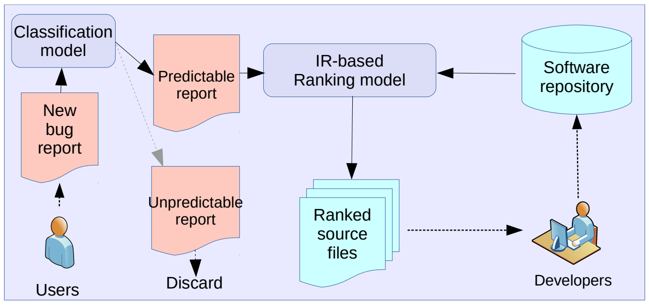

Figure 1: High level architecture: pre-filtering before ranking.

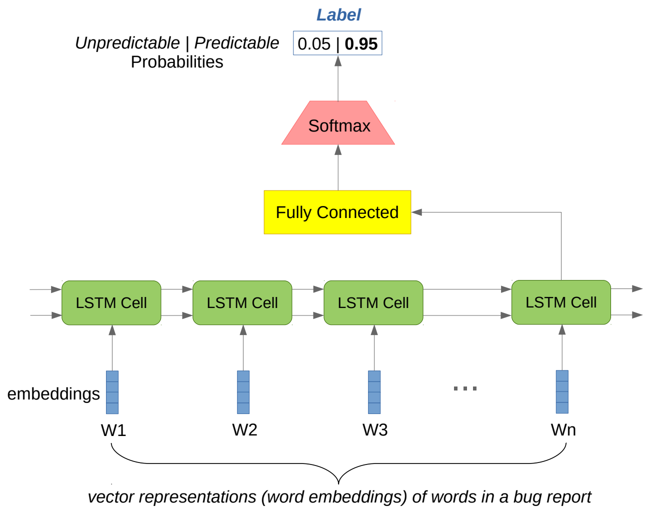
Label
Unpredictable | Predictable Probabilities
0.05 | 0.95
Softmax
Fully Connected
LSTM Cell
LSTM Cell
LSTM Cell
LSTM Cell
embeddings
W1
W2
W3
...
Wn
vector representations (word embeddings) of words in a bug report

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 [37]. We use Mikolov's Skip-gram model [34] 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*) [20]. 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 [18, 53]. 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.

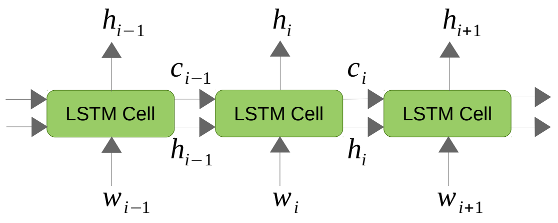

Figure 3: An LSTM Network.

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) [44] 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 [17], $\mathbf{b} \in \mathbb{R}^n$ 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 [29].

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

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

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

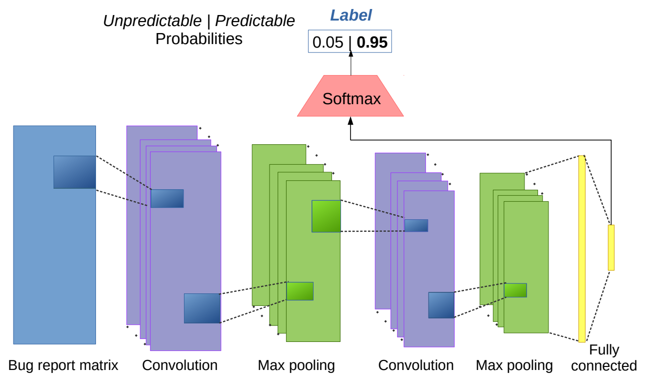

Figure 4: Bug-report classification architecture: using CNN.

training efficiency, helps avoid local minima, and achieves better convergence [18].

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 [16] 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) [36]. 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) [50].

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 *sigmoid* the activation function. 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 [22].

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

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

Table 1: Benchmark Projects: bug reports are split into a testing, a validation and a training set.

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

is general enough, as a comparison baseline, to work on any types of reports.

5 EVALUATION

This section discusses a set of extensive experiments to evaluate the effectiveness of the LSTM-based bug report classification approach on the bug locating task. Generally, we want to see whether this approach is able to filter out unhelpful bug reports to an IR-based system, whether it helps improve the bug-locating performance, what is the trade-off between precision and recall, and what is the difference in terms of performance between using different classification models.

5.1 IR-based Bug Locating System

We use a recent bug locating system provided by Ye et al. [58] in our experiments because: 1) it is an IR-based system; 2) it can be acquired; 3) it provides comparably good results with other state-of-the-art systems [30, 31]. This system uses a ***L**earning-to-**R**ank* technique combining **W**ord-**E**mbedding-based features and VSM-based features for bug locating. We denote it as LRWE in this paper.

5.2 Benchmark Datasets

We use the same benchmark dataset with [13, 30, 31, 57, 58] as shown in Table 1. Totally 9,105 bug reports and the corresponding source code packages from Eclipse Platform UI, SWT, and JDT are checkout and used in our experiments. We split these bug reports into a training dataset for training the classification models, a validation dataset for checking whether the models are converged or not, and a testing dataset for providing the testing result.

5.3 Labeling the Data

Data labeling provides the ground truth for the experiments. The goal of our bug report classification is to filter out unhelpful bug reports while keeping the helpful ones to an IR-based bug locating system for improving locating performance and reducing false positives. So the assessment of a bug report's helpfulness ("preictable" or "unpredictable"), instead of being done manually, should be performed automatically by the IR-based system used in the experiments.

We perform data labeling by running the chosen bug locating system LRWE for all the bug reports. More specifically, for every bug report, we run LRWE on the corresponding source code package checked out from a commit right before its fix commit. The output of the system is a ranked list of all the source code files, from which we label the bug report as "precitable" to the system if at least one buggy file can be seen within the top N positions. Otherwise, if the top N files in the ranked list are all irrelevant files, we label the bug report as "unpredictable". According to Miller's 7 ± 2 law [41], humans are able to handle seven plus or minus two tasks simultaneously. So we choose $N = 10$ under the assumption that a ranked list is useful to users if a real bug locates within the top 10 recommendations.

The output of data labeling is a set of 9,105 bug reports labeled as either "predictable" or "unpredictable" as the ground truth.

5.4 Evaluation Metrics

We evaluate our bug report classification approach from two perspectives. First, we want to test its classification performance. Second, we want to test if it can help improve the IR-based bug locating system's ranking performance. So we run experiments using the following evaluation metrics:

- *true positive*: a "predictable" bug report that is also classified as "predictable"
- *false positive*: an "unpredictable" bug report that is classified as "predictable"
- *true negative*: an "unpredictable" bug report that is also classified as "unpredictable"
- *false negative*: a "predictable" bug report that is classified as "unpredictable"
- ***Accuracy***: a standard metric measuring the correctness of the classification results.

$$Accuracy = \frac{|\text{true positives}| + |\text{true nagatives}|}{\text{the totaly number of instances}} \tag{5}$$

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

$$Precision = \frac{|\text{true positives}|}{|\text{true positives}| + |\text{false positives}|} \tag{6}$$

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

$$Recall = \frac{|\text{true positives}|}{|\text{true positives}| + |\text{false negatives}|} \tag{7}$$

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

$$F\text{-}Measure = (1 + \beta) \cdot \frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{8}$$

  When we give equal weights to Precision and Recall by setting $\beta = 1$, we have **F1-Measure** that is called the harmonic mean of Precision and Recall.
  When we give more weights to Precision by setting $\beta = 0.5$, we have **F0.5-Measure** that considers Precision is more important.
- *Mean Average Precision (**MAP**)*: a standard metric measuring the overall ranking performance of an IR system [40]. It is defined as the mean of the average precision over all queries.

MAP is widely used in measuring the ranking performance of IR-based bug locating systems [24, 25, 30, 51, 57, 58, 61].

- *Mean Reciprocal Rank (**MRR**)*: a metric measuring the ranking performance of an IR system on the first recommendation [54].

We use Accuracy, Precision, Recall, F1-Measure and F0.5-Measure to evaluate the bug report classification results. We use MAP and MRR to measure the IR system's ranking results.

5.5 Training and Testing Procedure

As discussed in Section 3.4, the models are trained over a maximum 500 epochs with an early stopping criterion.

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

Figure 6: The training and testing procedure.

Figure 6 shows the details of our training and testing procedure. When the models are trained over every epoch, we test the models on the validation dataset and keep track of the performance. If a continuous decrease of the validation Accuracy is observed over 10 times, we assume the models are converged and stop the training process. At the end, we report the results on the testing dataset when the models produce the best validation Accuracy.

5.6 Tuning the Hyperparameters

We tune models' hyperparameters on the validation dataset using the the same procedure as shown in Figure 6 but without testing on the testing set.

For LSTM-network, the number of hidden units in the memory cell is set to 32, the dropout rate on the input layer is 0.9, the output dropout rate is 0.7, and the learning rate 0.003.

For CNN, the first convolutional layer uses eight filters with size 5x5. The second convolutional layer uses sixteen 5x5 filters. The pooling size is 2x2. The fan-out of the first fully connected layer is 512.

For multilayer perceptron, all three hidden layers use 500 internal nodes.

The learning rates for both CNN and perceptron are 0.001. The training batch size for these three models are all set to 64.

The length thresholds of the simple baseline approach are chosen based on its performance on the validation set as well. Specifically, we set the thresholds to be 7,6,7 for the three benchmark projects respectively.

5.7 Results and Analysis

The rest of this section reports results to answer the following research questions.

RQ1: Can the LSTM-network approach, compared with using other classification models, helps filter out "unpredictable" bug reports?

RQ2: Can the LSTM-network approach helps improve the ranking performance of the IR-based system on bug locating?

RQ3: Can we futher increase the Precision? If so, what is the trade-of betwen Recall?

5.7.1 ***RQ1:*** *Can the LSTM-network approach, compared with using other classification models, helps filter out "unpredictable" bug reports?*

Table 2: Classification results of using difference models: MLP refers to the multilayer perceptron model, SB refers to the simple baseline approach, and NC (No Classification) refers to classifying all the instances (bug reports) as positive ("predictable").

Project	Metric	LSTM	CNN	MLP	SB	NC
Eclipse Platform UI	Accuracy	**0.670**	0.650	0.645	0.658	0.658
	Precision	**0.703**	0.672	0.676	0.672	0.658
	Recall	0.862	0.901	0.884	0.939	**1**
	F1-Measure	0.775	0.770	0.766	0.783	**0.794**
	F0.5-Measure	**0.730**	0.708	0.709	0.712	0.706
SWT	Accuracy	**0.694**	0.685	0.692	0.659	0.692
	Precision	0.694	0.698	0.692	**0.700**	0.692
	Recall	**1**	0.963	0.884	0.887	**1**
	F1-Measure	**0.819**	0.809	0.783	0.783	0.818
	F0.5-Measure	**0.739**	0.738	0.738	0.731	0.738
JDT	Accuracy	**0.763**	0.747	0.710	0.737	0.759
	Precision	**0.762**	0.761	0.758	0.761	0.759
	Recall	**1**	0.971	0.908	0.952	**1**
	F1-Measure	**0.865**	0.853	0.823	0.846	**0.863**
	F0.5-Measure	**0.8**	0.796	0.784	0.793	0.797

To answer the first research question, we run different classification models on three benchmark projects. The results are shown in Table 2, where each column shows the results of a model on different projects.

We obtain the following observations from the results: (1) Compared with not doing bug report classificationusing, using the LSTM-network approach helps (as Precision increases) filter out "unpredictable" reports. (2) While the LSTM-network increases the classification precision, it drops the recall. It helps reduce false positives but also reduce true positives. (3) Using LSTM-network can increase F0.5-Measure but may decrease F1-Measure. If we consider Precision and Recall are equally important, LSTM may not help. But if we prefer precision over recall under a certain trade-off (F0.5-Measure), using LSTM-network helps. (4) Using LSTM-network achieves better classification performance than using CNN, multilayer perceptron, and the simple baseline approach. It shows that the long-term dependencies learned by LSTM is useful to decide the usefulness of a bug report to IT-based bug locating. (5) Interestingly, the simple baseline approach provides comparable performance with CNN and multilayer perceptron. One potential reason may be because the content size of a report is more important than the locality dependencies of the bug report matrix. (6) We also observe that the performane difference between LSTM and other models on SWT and JDT, compared with on Eclipse, are marginal. One potential reason may be the smaller training size and testing size. But it still produces better precisions while even keeping the same F1-Measure and better F0.5-Measure on these two projects.

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

5.7.2 ***RQ2:*** *Can the LSTM-network approach helps improve the ranking performance of the IR-based system on bug locating?*

Table 3: Bug locating results: SB refers to the simple baseline approach, NC (No Classification) refers to classifying all the instances (bug reports) as positive ("predictable").

Project	Metric	LSTM	SB	NC
Eclipse Platform UI	MAP	**0.405**	0.382	0.369
	MRR	**0.460**	0.432	0.419
	Recall	0.862	0.939	**1**
	F0.5-Measure	**0.730**	0.712	0.706
SWT	MAP	0.383	**0.393**	0.382
	MRR	0.444	**0.455**	0.443
	Recall	**1**	0.887	**1**
	F0.5-Measure	**0.739**	0.731	0.738
JDT	MAP	**0.430**	0.425	0.425
	MRR	**0.521**	0.520	0.516
	Recall	**1**	0.952	**1**
	F0.5-Measure	**0.8**	0.793	0.797

To test whether our approach helps improve the IR-based system on bug locating, we apply our LSTM-based model on bug reports in the testing dataset and run the IR-beased bug locating system LRWE on only the "predictable" reports. Table 3 shows the comparison results with not doing bug report classification. We observed an obvioius performance improvement in terms of MAP and MRR over both the simple baseline approach and not doing classification on Eclipse Platform UI. For SWT the JDT, like in Table 3, the advantage is not obvious. It may caused by the smaller training and testing size. We leave it to future work to collect more data and perform more fine-grained tuning of these two projects. To save time, we do not run experiments on CNN and multilayer perceptron because they do not show comparable classification performance with LSTM in the preivious section.

Our answer to *RQ2* is that the LSTM-network approach, under the trade-off (F0.5-Measure) between recall, helps improve the IR-based bug location system's ranking performance.

5.7.3 ***RQ3:*** *Can we futher increase the Precision? If so, what is the trade-of between Recall?*

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

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

We run experiments using different classification models on the Eclipse Platform UI project by tuning the probability threshold from 0 to 1. For the simple baseline approach, we tune the length as the the threshold. Then we draw a learning curve for each evaluation metric. The learning curves are shown in Figure 7 for LSTM, Figure 8 for CNN, Figure 9 for multilayer perceptron, and Figure 10 for the simple baseline approach.

An overall observation from the results is that the precision increases when we increase the threshold, but the recall drops. When we give more preference to precision, we observe that the F0.5-Measure value also increase.

More specifically, take the LSTM result shown in Figure 7 for example, both Precision and F0.5-Measure increase with the probability threshold. When the threshold is set to 0, which means no classifications at all, the Precision is 0.658 and the F0.5-Measure is 0.706. After we increase the threshold to 0.5, we obtain the result shown in Table 2, where Precision is 0.703 and F0.5-Measure is 0.73. Then we continue to increase the probability threshold. When the threshold increases to 0.8, which means that we classify a bug report as "predictable" only when $P(\text{“}predictable'') > 0.8$, the Precision also increases from 0.703 to 0.716, and the F0.5-Measure increases from 0.73 to 0.735.

Next, to get more insights into the difference between models, we compare the changes of precision over the changes of recall in Figure 11, from which we clearly observe that the LSTM-network (RNN) model achieves better performance in terms of Precision than other models before the Recall drops too much. We also observe that CNN gives better Precision when Recall drops below 0.5 that is too low to be useful.

In summary, we answer *RQ3* that we can further increase the Precision while keeping a certain trade-off (in terms of F0.5-Measure) between recall. The LSTM-network achieves the best trade-off between Precision and Recall.

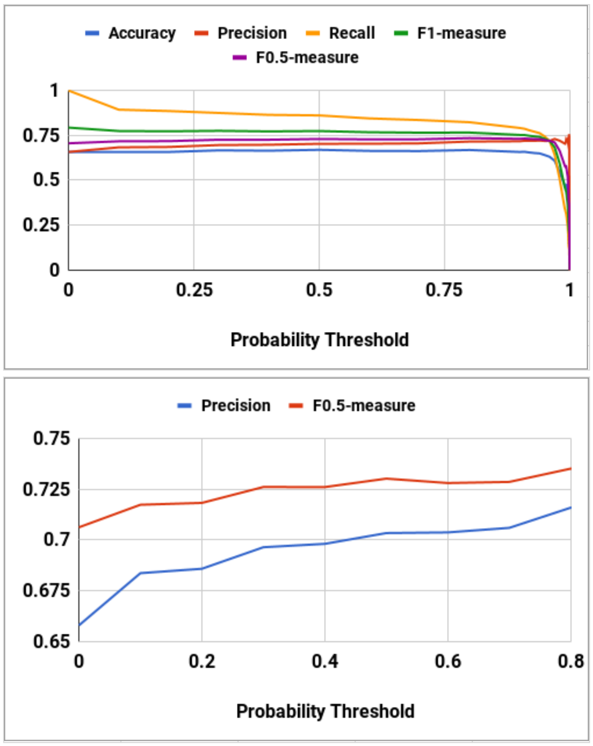

Figure 7: Learning curves for LSTM.

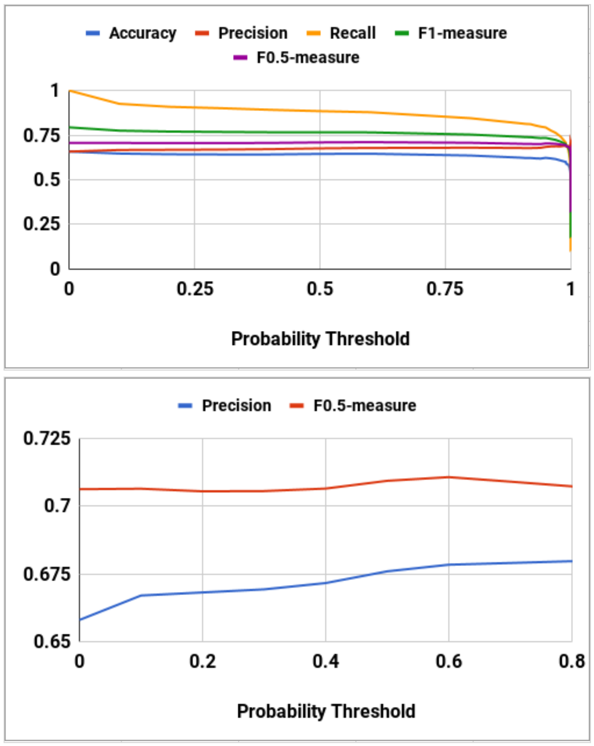

Figure 9: Learning curves for multilayer perceptron.

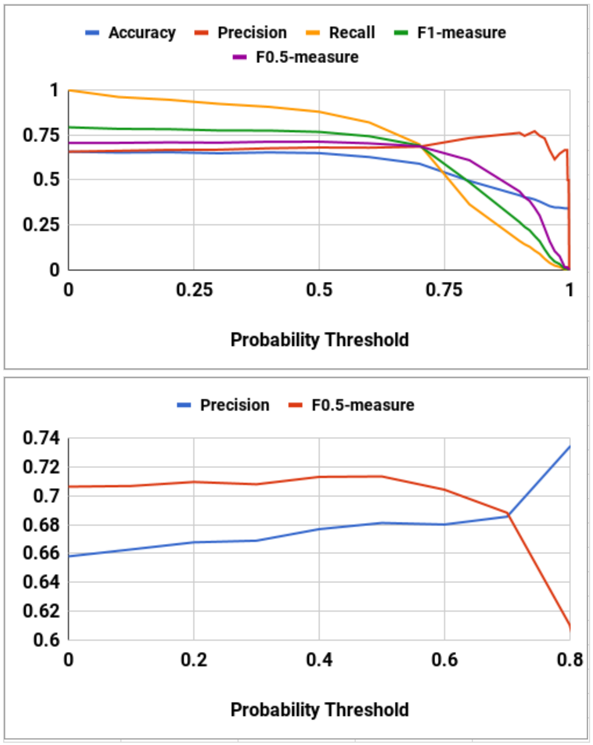

Figure 8: Learning curves for CNN.

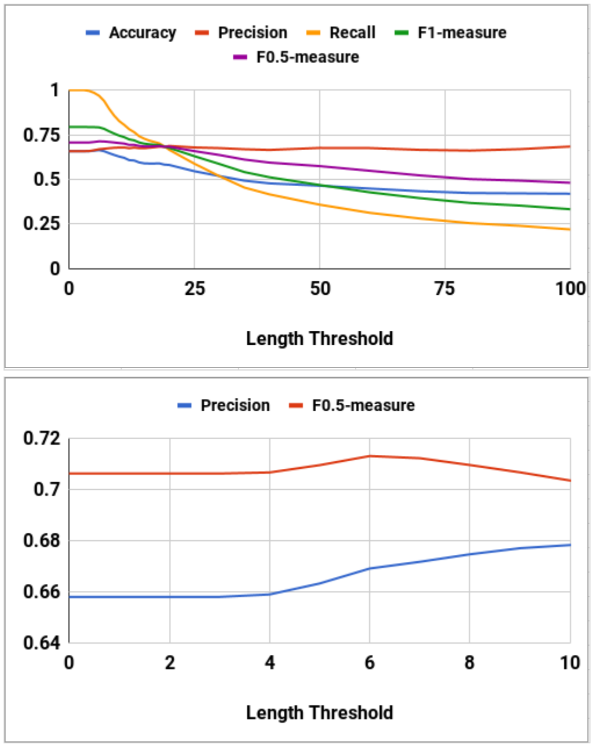

Figure 10: Learning curves for the simple baseline approach.

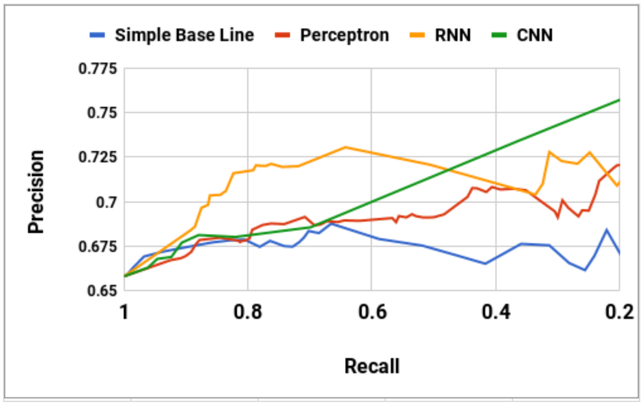

Figure 11: Precision vs. Recall.

6 RELATED WORK

In this section, we describe work in other areas of IR-based bug report handling, other uses of neural networks in software engineering, and briefly touch on non-IR-based approaches to bug report handling.

6.1 Bug Report Handling

Lam et al. [30] seeks to improve bug report handling by automating the task of associating buggy files with a bug report. In order to overcome the lexical mismatch problem of the natural language used in bug reports not matching the terms and code tokens in source files, they combined rSVM information retrieval with deep neural networks to associate terms in bug reports to terms in source files. Their resulting model, DnnLoc, is able to suggest likely source code files that contain the bug described in a bug report.

Huo et al. [24, 25] propose a couple of approaches to localize buggy source files from a bug report. They first propose a novel convolutional neural network NP-CNN that leverages the structural information of source code in addition to the lexical information to accomplish this task. They follow with another model LS-CNN that combines CNN and LSTM to additionally utilize the sequential information of source code.

Ye et al. [57, 58] develops a learning-to-rank model to combine various features for ranking source files for bug reports. The model is trained using source code contents, API descriptions of the code, bug-fixing history, and the code change history information of previously solved bug reports. Further work to bridge the lexical gap between bug reports and source files was done using word embeddings to train a model to estimate semantic similarities between bug reports, source code, and API/reference documents.

Zhou et al. [61] implemented BugLocator that locates files based on ranking by textual similarity of bug reports and source code using a revised Vector Space Model (rSVM). Sahar et al. [51] outperforms BugLocator with BLUiR that uses structural information of code to enable more accurate bug localization.

Kim et al. [28] apply Naïve Bayes to localize source code files for a bug report based on previously fixed files as labels. In order to improve localization accuracy, they add an initial phase where bug reports are classified to predictable or deficient based on prediction history of resolved bug reports. Deficient bug reports are not localized to code to avoid misleading recommendations.

Lukins et al. [39] use a Latent Dirichlet Allocation (LDA) based technique that is accurate and scaleable for automatic bug localization. Nguyen et al. [45] uses the shared technical aspects in the text of bug reports and corresponding source code to implement a LDA based system BugScout to correlate reports to buggy code.

Rao et al. [49] compared Unigram Model, Vector Space Model, Latent Semantic Analysis Model, Latent Dirichlet Allocation Model, and Cluster Based Document Model for bug localization. They found that more sophisticated models (LDA, LSA, CBDM) did not outperform simpler text models (UM, VSM).

Determining the severity of bug reports automatically is another area where handling of bug reports can be improved. Lamkanfi et at. [32] use a Naïve Bayes based approach to investigate if severity can be accurately predicted. They conclude that a sufficient training set can achieve reasonable prediction accuracy. Zhang et al. [59] describe a system to find similar historical bug reports utilizing a modified REP algorithm and K-Nearest Neighbor. Then, an improved performance severity predicition algorithm is developed with the extracted features of the bug reports.

Another direction for reducing effort of handling bug reports is to automate triage of bug reports to developer(s) that are likely to resolve them. Anvik et al. [1, 2] used support vector machines and other machine learning approaches to implement developer recommending models achieving varying degrees of precision. Hu et al. [23] implement recommendation method called Bug Fixer that utilizes historic information of source code components where developers have have fixed bugs previously. Zhang et al. [60] implement a hybrid system that utilizes unigram model to find similar bug reports and then recommends a developer based on developer's probability to fix and a model of developer's activity and experience. Bhattacharya et al. [7] employ a set of machine learning tools and tossing graphs to accurately assign bugs to developers. Xuan et al. [56] use a model of instance selection and feature selection determined by historic bug data sets to reduce data scale and improve accuracy of bug triage. Shokripour et al. [52] uses an approach that uses noun extraction and simple term weighting to predict bug location and then uses a location-based approach to recommend assigment of the bug to a developer.

6.2 Using Neural Networks to Support Software Engineering

Effort estimation is necessary for planning and managing a software project. Choetkiertikul et al. [11] utilizes deep learning with long short-term memory and recurrent highway network to facilitate effort estimation for agile projects. They use deep learning to model and predict estimations of story points, a unit of measure for the effort to complete a user story or resolve an issue.

Developers often need to utilize APIs to implement functionality, but it can be a significant obstacle to deal with unfamiliar libraries or frameworks. Gu et al. [19] utilize RNN Encoder-Decoder for a deep learning approach called DeepAPI. DeepAPI allows a natural language query to accurately generate a relevant API sequence.

Online developer forums are full of individual units of programmer knowledge that have potential to be linked for being related, duplicates, etc. Xu et al. [55] utilize word embeddings and convolutional neural networks for a deep learning based approach to semantically linking knowledge units on StackOverflow that outperforms traditional methods. Fu et al. [15] follow up this approach with a differential evolution approach that achieve similar results on the scale of minutes rather than hours with the deep learning approach. They show that deep learning may provide benefits for software engineering but simpler or faster methods should still be considered.

6.3 Non-IR-Based Bug Report Handling

Information retrieval approachs are not the only way to try handling bug reports. There are approaches that are not IR-based or augment/combine with IR to accomplish bug report handling tasks. Cleve et al. [12] focus on cause transitions to find locations of defects. Dit et al. [14] utilizes web mining algorithms to analyze execution information. Poshyvanyk et al. [47, 48] have utilized both Formal Concept Analysis and scenario-based probabilistic ranking of events. Liu et al. [38] uses a model based on pattern evaluation between correct and incorrect runs to quantify bug-relevance. Jin et al. [26] used synthesized passing and failing executions to perform fault localization. Le et al [3, 35] utilizes approaches of program spectra analysis to find suspicous words and invariant mining. Jones et al. [27] implements Tarantula approach of generating likelihood/suspicion for each statement of source code using the code entities executed by passing and failing test cases.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces using a LSTM-network for classifying a bug report as either “predictable” or “unpredictable”. While a “pedictable” report is considered helpful to an IR-based bug locating system to find the bug, an “unpredictable” report is deemed unhelpful to an IR system and is discarded. Evaluation results show that our classification model helps filter out “unpredictable” reports and improves the ranking performance of a state-of-the-art IR system on software bug locating. Among different classification models, the LSTM-network achieves the best trade-off between precision and recall. While our approach helps for IR-based bug locating, we observe the decrease of F1-Measure and the increase of F0.5-Measure. So we conclude that if precision and recall are given equal preference, our classification model is not helpful. But in the situation that precision is more important than recall, our classification model helps while keeping a certain trade-off between recall.

In future work, we will test the effectiveness of our approach using more IR-based bug locating systems on more software projects. In parallel, We plan to explore alternative methods like *sent2vec* [46] for converting bug report into vector representations. Beside, we also plan to manualy review different bug reports to get insights of what type of quality make a report to be helpful to an IR system. We plan to create features that effectively represent the quality of bug reports and combine them with neural-network-based features for more precise classiication.

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. New York, NY, USA, 361–370.

[2] John Anvik and Gail C. Murphy. 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.* 20 (2011), 10:1–10:35.

[3] 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

[4] 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

[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] Pamela Bhattacharya, Iulian Neamtiu, and Christian R. Shelton. 2012. Automated, Highly-accurate, Bug Assignment Using Machine Learning and Tossing Graphs. *J. Syst. Softw.* 85, 10 (Oct. 2012), 2275–2292. https://doi.org/10.1016/j.jss.2012.04.053

[8] 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.

[9] 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.

[10] 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.

[11] 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

[12] 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.

[13] T. Dilshener, M. Wermelinger, and Y. Yu. 2016. Locating Bugs without Looking Back. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 286–290. https://doi.org/10.1109/MSR.2016.037

[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] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/3106237.3106256

[16] 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

[17] 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

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

[19] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[20] 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

[21] 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.

[22] 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

[23] Hao Hu, Hongyu Zhang, Jifeng Xuan, and Weigang Sun. 2014. Effective Bug Triage Based on Historical Bug-Fix Information. *2014 IEEE 25th International Symposium on Software Reliability Engineering* (2014), 122–132.

[24] 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

[25] 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

[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] 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.

[28] 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.

[29] 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

[30] 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

[31] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 218–229. https://doi.org/10.1109/ICPC.2017.24

[32] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. 2010. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 1–10. https://doi.org/10.1109/MSR.2010.5463284

[33] 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.

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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.

[39] 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.

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

[41] George A. Miller. 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* 63, 2 (March 1956), 81–97.

[42] 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

[43] 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

[44] 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

[45] 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

[46] M. Pagliardini, P. Gupta, and M. Jaggi. 2017. Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features. *ArXiv e-prints* (March 2017). arXiv:cs.CL/1703.02507

[47] 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.

[48] 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.

[49] 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.

[50] 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

[51] 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.

[52] Ramin Shokripour, John Anvik, Zarinah M. Kasirun, and Sima Zamani. 2013. Why So Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. Piscataway, NJ, USA, 2–11. http://dl.acm.org/citation.cfm?id=2487085.2487089

[53] 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

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

[55] 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

[56] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Transactions on Knowledge and Data Engineering* 27 (2015), 264–280.

[57] 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

[58] 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

[59] Tao Zhang, Jiachi Chen, Geunseok Yang, Byungjeong Lee, and Xiapu Luo. 2016. Towards More Accurate Severity Prediction and Fixer Recommendation of Software Bugs. *J. Syst. Softw.* 117, C (July 2016), 166–184. https://doi.org/10.1016/j.jss.2016.02.034

[60] Tao Zhang and Byungjeong Lee. 2013. A hybrid bug triage algorithm for developer recommendation. In *SAC*.

[61] 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