Deep Just-In-Time Defect Localization

Fangcheng Qiu, Zhipeng Gao, Xin Xia, David Lo, John Grundy and Xinyu Wang

Abstract—During software development and maintenance, defect localization is an essential part of software quality assurance. Even though different techniques have been proposed for defect localization, i.e., information retrieval (IR)-based techniques and spectrum-based techniques, they can only work after the defect has been exposed, which can be too late and costly to adapt to the newly introduced bugs in the daily development. There are also many JIT defect prediction tools that have been proposed to predict the buggy commit. But these tools do not locate the suspicious buggy positions in the buggy commit. To assist developers to detect bugs in time and avoid introducing them, just-in-time (JIT) bug localization techniques have been proposed, which is targeting to locate suspicious buggy code after a change commit has been submitted. In this paper, we propose a novel JIT defect localization approach, named DEEPDL (Deep Learning-based defect localization), to locate defect code lines within a defect introducing change. DEEPDL employs a neural language model to capture the semantics of the code lines, in this way, the naturalness of each code line can be learned and converted to a suspiciousness score. The core of our DEEPDL is a deep learning-based neural language model. We train the neural language model with previous snapshots (history versions) of a project so that it can calculate the naturalness of a piece of code. In its application, for a given new code change, DEEPDL automatically assigns a suspiciousness score to each code line and sorts these code lines in descending order of this score. The code lines at the top of the list are considered as potential defect locations. Our tool can assist developers efficiently check buggy lines at an early stage, which is able to reduce the risk of introducing bugs in time and improve the developers' confidence in the reliability of their software. We conducted an extensive experiment on 14 open source Java projects with a total of 11,615 buggy changes. We evaluate the experimental results considering four evaluation metrics. The experimental results show that our method outperforms the state-of-the-art by a substantial margin.

1 INTRODUCTION

In software development and maintenance, developers often spend much effort and resources for program debugging. For example, software debugging can cost 80% of the total software cost for some software projects [1]. Nevertheless, identifying the locations of bugs has historically been a manual task, which is considered to be time-consuming and labor-intensive [2]. In this study, our research aims to help developers to reduce the manual efforts regarding the software debugging process. Two types of software engineering tasks are relevant to our work: Just-In-Time (JIT) defect prediction and fault localization. (i) **Fault localization**: this task aims to help developers localize potential faulty code elements (e.g., statements or methods) by analyzing various dynamic execution information (e.g., failed/passed tests, bug reports). Previous work investigated the fault localization task by using information retrieval (IR) based techniques [3], spectrum-based techniques [4], or learning based techniques [5], [6]. However, one of the crucial disadvantages of these fault localization techniques is that they heavily depend on the dynamic execution information and only work after the defect has been exposed, which can be too late and costly for the newly introducing bugs. Besides, spectra-based techniques require test cases that are often unavailable [7]–[9]. IR-based does not work until line level (usually only until file/method level). (ii) **JIT defect prediction task**: for a given commit, the JIT defect prediction tool aims to help developers to check if the commit is a buggy commit. Although different techniques have been proposed to predict the buggy commit just-in-time [10], [11], these prior works do not locate the suspicious positions. Considering a submitted commit usually involves dozens of changed files with hundreds of added lines (e.g., according to our empirical study, the average number of added lines of a commit is 98), finding the buggy line from a set of irrelevant lines is still tedious and time-consuming.

To address the above challenges regarding the fault localization and JIT defect prediction task, Yan et al. [12] first proposed the task of "**Just-in-time (JIT) defect localization**", which aims to locate buggy code elements before the defect symptoms have cased any negative effects. Compared with the task of JIT defect prediction and fault localization, JIT defect localization can yield the following benefits: (i) Fine-granularity detection. Compared with JIT defect prediction which detects the buggy changes at file-level or module level, JIT defect localization can locate the buggy code elements at a fine-granularity (i.e., line-level). Such fine-granularity detection can save the developer's time and effort to locate and address the defects. (ii) Early stage detection. Compared with fault localization, which heavily relies on defect symptoms and can only work after the defects have been exposed, JIT defect localization is performed when code change happens, in other words, JIT defect localization locates the buggy code lines whenever a commit is submitted. The early stage detection can prevent the buggy code at an early stage and give developers

- *Fangcheng Qiu and Xinyu Wang are with the College of Computer Science and Technology, Zhejiang University, China.*
  E-mail: {fangchengq, wangxinyu}@zju.edu.cn
- *Zhipeng Gao and John Grundy are with the Faculty of Information Technology, Monash University, Melbourne, Australia.*
  E-mail: {zhipeng.gao, john.grundy}@monash.edu
- *Xin xia is with Software Engineering Application Technology Lab, Huawei, China.*
  E-mail: xin.xia@acm.org
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Xin Xia is the corresponding author.*

immediate feedback.

Yan et al. [12] developed their JIT defect localization framework based on the basic idea of "software naturalness". Hindle et al. [13] have investigated the possibility of using "naturalness" for the defect prediction task, because buggy code tends to be more "unnatural" compared with correct code [14]. They built a traditional language model using n-gram techniques to estimate the "naturalness" of a submitted change, However, their approach still suffers from several inherent disadvantages.

- *Contextual Features.* Their approach employed n-gram techniques for calculating naturalness scores, considering that n-gram technique is based on bag-of-words (BOW) models, which can only capture the lexical level features. When developers write code, the code line is not written as an isolated element, developers consider the connection of each code line with respect to its context. Capturing the semantic level features and contextual relations between the code lines can boost the model.
- *Out-of-Vocabulary (OOV) Problem.* If a word appears in the testing set but not in the training set, a traditional model treats this word as an unknown word and fails to predict it in the testing phase. The OOV problem occurs very frequently in practice because different developers tend to define variables according to their own habits. Previous studies [15] suggest that such OOV problems may greatly hinder the learning performance of the model. Their approach can hardly handle the tokens out of vocabulary.

To address the above challenges, in this paper, we propose a novel approach, named DEEPDL (**Deep** Learning-based Just-in-Time **D**efect **L**ocalization), that can help developers to locate the buggy code lines at check-in time (the inspection phase that after developers change source code, before running the program) efficiently and accurately. DEEPDL consists of three stages: data processing, model training and model application. Particularly, in the data processing stage, we collect code lines from the latest snapshot of a software project as training samples to train a neural language model. To alleviate the OOV problems, we leverage a Byte-Pair Encoding (BPE) algorithm [16] to tokenize source code, which can greatly reduce the size of the source code vocabulary and successfully solve the unseen word in the testing set. DEEPDL can then be trained with these training samples. During the model training stage, to effectively capture the contextual features of the code lines and their relations, we leverage a neural language model to learn the naturalness of the code lines. Our neural language model takes a sequence of code line blocks as input and outputs a code line sequence, which can be formulated as a sequence-to-sequence learning problem.

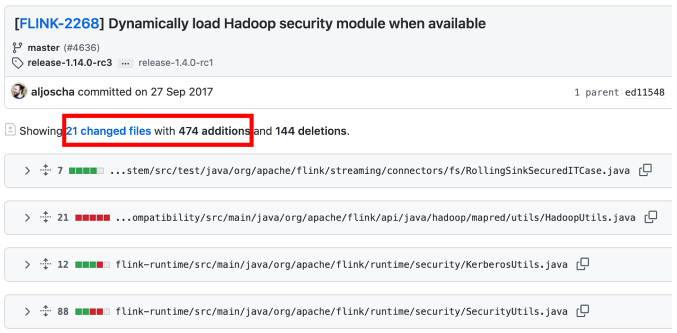
When it comes to the model application stage, when a developer submits a new code change, after going through the same data processing procedures, the newly changed code is analysed by DEEPDL to estimate its "suspiciousness" score. The code line with the highest suspiciousness score is considered to be a possible defect location. DEEPDL can be used to assist developers in identifying the location of the potential buggy code lines during code changes, and can consequently reduce or even avoid the introduction of bugs in daily development.

To demonstrate the effectiveness of our approach, we train and evaluate DEEPDL on a Java dataset which contains 14 open source Java projects from GitHub. We use the source code from previous snapshots of the project to train the model and use the buggy changes introduced after this snapshot to evaluate the model. We measure the performance of DEEPDL using Top-k accuracy, MRR and MAP. The experimental results demonstrate that DEEPDL achieves a Top-1 accuracy 0.32, Top-5 accuracy 0.59, MRR 0.44, and MAP 0.40 on average, outperforming the state-of-the-art approachs by Yan et al.'s approach [12] and CC2Vec [17].

We make the following key contributions with this work:

- We propose the first neural language model, DEEPDL, for just-in-time line-level defect localization task. Our model can help developers locate the suspicious bug code lines in a bug introducing change.
- We perform extensive experiments on DEEPDL and our results demonstrate the effectiveness and superiority of our solution wrt. the existing work.
- We confirm that a large training corpus makes a cross-project model achieve comparable performance to a within-project model.

The organization of this paper is as follows. Section 2 describes the background of the language model and sequence-to-sequence model. Section 3 describes the detailed design of our approach. Section 4 describes our experimental design. Section 5 presents the evaluation results. Section 6 discusses our work and gives the threats to validity. Section 7 presents the related work. We conclude the paper in Section 8.



[FLINK-2268] Dynamically load Hadoop security module when available

master (#4636)
release-1.14.0-rc3 ... release-1.4.0-rc1

aljoscha committed on 27 Sep 2017 1 parent ed11548

Showing 21 changed files with 474 additions and 144 deletions.

```
> 7 ...stem/src/test/java/org/apache/flink/streaming/connectors/fs/RollingSinkSecuredITCase.java
> 21 ...ompatibility/src/main/java/org/apache/flink/api/java/hadoop/mapred/utils/HadoopUtils.java
> 12 flink-runtime/src/main/java/org/apache/flink/runtime/security/KerberosUtils.java
> 88 flink-runtime/src/main/java/org/apache/flink/runtime/security/SecurityUtils.java
```

Fig. 1: An Example Commit in Flink

2 MOTIVATION

Figure 1 shows a commit example from the Flink project, where the developer has submitted a commit for the purpose of "*Dynamically load Hadoop security module when available*", this single commit involves 21 changed files with 474 additions and 144 deletions. Even the state-of-the-art JIT defect prediction tool can successfully identify whether this commit is buggy or not, manually checking the changed files one by one within this commit is still time-consuming and labor-intensive. Therefore it is preferable to have a tool that can check the potential defective lines within these

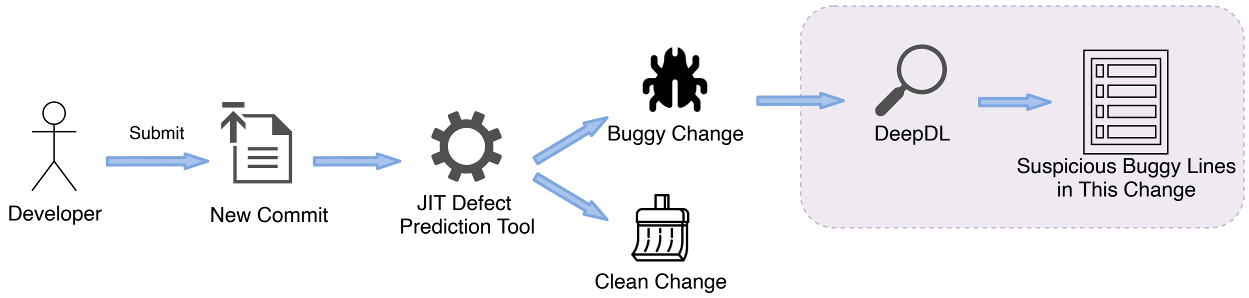

Developer
Submit
New Commit
JIT Defect Prediction Tool
Buggy Change
Clean Change
DeepDL
Suspicious Buggy Lines in This Change

Fig. 2: Usage Scenario of DEEPDL

large number of changed files automatically, We illustrate some key usage scenarios of our proposed tool, DEEPDL, as follows: (i) First of all, DEEPDL is able to quickly identify the locations of the defects for the historical commits with the help of JIT defect prediction tool. For example, as shown in Figure 2, developers can first leverage the JIT defect prediction tool proposed by Hoang et.al [17] to check the buggy commits, then our tool DEEPDL can be used to automatically pinpoint the suspicious buggy line among a set of non-buggy lines. The developers can thus focus on the reported bugs instead of painstakingly browsing the changed files one by one. (ii) Secondly, our tool can also be used to remind developers in identifying the potential defect localization when submitting a new commit. When developers submit a code change, DEEPDL can automatically locate the suspicious buggy code and provide the notifications, therefore the DEEPDL can assist developers to reduce the risk of introducing bugs and improve the software's reliability.

3 BACKGROUND

Our work adopts several recent advanced techniques from natural language processing and deep learning [14], [18]–[20], in this section, we presents the background of these key related techniques.

3.1 Language Model

Our work is inspired by the idea of language model used in the Natural Language Processing (NLP) field. To adapt this idea to our task of defect localization, we want to build a language model to estimate the "software naturalness" for a given code fragment. Because compared with buggy code, the clean code tends to be more "natural".

3.1.1 Traditional Language Model

Traditional language model is a probability distribution over sequences of words. Given a sequence of tokens $w = [t_1, t_2, ...t_i]$, the language model estimates the probability of it. The probability is computed as:

$$P(w) = P(t_1) \prod_{i=2}^{n} P(t_i|t_1, ..., t_{i-1}) \quad (1)$$

$P(t_i|t_1, ..., t_{i-1})$ denotes the probability that token t_i follows the previous tokens, i.e., $t_1, ..., t_{i-1}$. This traditional language model predicts the next word by looking up the history of words. As a result, the language model assigns a probability (or a score) to a sequence of words. In the work of Yan et al. [12], they adopted a traditional N-gram language model to calculate the "naturalness" score of a code fragment. The higher score of a new code fragment is, the more natural the new code fragment is with the training code corpus.

3.1.2 Neural Language Model

The traditional language model can only capture the lexical level features, most recently, deep neural networks have been introduced to build the neural language model (NLM), which can improve the traditional language model. Mikolov et al. [21] first proposed a neural language model based on Recurrent Neural Network (RNN), since RNN is originally designed for sequences and can catch the chain-like natures. Sundermeyer et al. [22] introduced Long short-term memory (LSTM) neurons into neural language model and proposed, which aims to address the long-term dependency problem which can not be solved by the RNN language model. However, the LSTM language model is unidirectional that only predicts the outputs from past inputs. A bidirectional RNN model [23] utilizes past and future contexts by processing the input data in both directions. Bidirectional LSTM help us estimate the probability by using the left and right context of that word. Bidirectional LSTM using past and future contexts has achieved improvements. To better capture the relationship of the current word and its context, an attention mechanism is also added to language model. Tran et al. [24] and Mei et al. [20] demonstrated that an attention mechanism can improve the performance of RNN language models. The neural language models have been shown to outperform n-gram based language models, however, they are unable to handle the subword information. This is especially problematic in dealing with rare words or domains with dynamic vocabularies. To the best of our knowledge, our work is the first to employ a neural language model for just-in-time defect localization tasks.

3.1.3 Naturalness of Code

In general, naturalness represents how "surprised" an element is by the given document. The naturalness of code was first proposed by Hindle et al. [13]. They found that the source code is also very repetitive even more so than natural languages and the repetitiveness of source code can be captured by language models. The naturalness of code is widely used to detect bugs or syntax errors. Ray et al. [14] focused on the "naturalness" of buggy code and found that buggy code tends to be more unnatural than the clean code. And Santos et al. [25] proposed a tool to detect and correct the syntax errors based on naturalness of code. Based on the code naturalness, Yan et al. [12] proposed a two-phase framework to detect buggy commits and localize buggy code lines in buggy commits. Based on their findings, we also leverage naturalness in our approach.

3.2 Seq2Seq Model

The language model is a fundamental task in natural language processing, which is formalized as a probability distribution over a sequence of target words. The language model has various applications (e.g., speech recognition, text generation and machine translation), all these applications can be viewed as generating a variable-length sequence of tokens from a variable-length sequence of input data. Intuitively, the sequence-to-sequence (Seq2Seq) models can model these mappings well and achieve state-of-the-art results with respect to the aforementioned applications [26]–[28]. Besides, both the encoder and decoder of Seq2Seq model can be trained with paired text to obtain as language model [29], [30]. Similarly, for our task of JIT defect localization, we aim to learn the naturalness between the newly added line with respect to its surrounding lines, we thus adopt the Seq2Seq model to train a neural language model (i.e., the input sequence is a code block and the output sequence is the code line). Ideally, our neural language model will take a code block as input and generate a "clean" code line as output with respect to the code block. Then a "naturalness" score can be calculated (measured by entropy) between the added line and the generated "clean" code line.

3.2.1 Encoders & Decoders.

In general, a Seq2Seq model uses an encoder-decoder architecture. It first employs an encoder to map the input sequence into a fixed dimensional vector, then this vector is used by the decoder to decode the target sequence. **Encoder** is responsible for embedding the input sequence into a contextualized hidden state vector. Particularly, given the input sequence $\mathbf{X} = (x_1, x_2, ..., x_n)$ comprising a number of n tokens. These tokens are fed sequentially into the the encoder, which generates a sequence of hidden states $\mathbf{H} = (h_1, h_2, ..., h_n)$. The final hidden state h_n can be used as the embedding vector v of the whole input sequence. **Decoder** is responsible for generating the target sequence based on the embedding vector. Specifically, at time step t, the decoder takes the embedding vector of the previous word y_{t-1} and the previous hidden state s_{t-1} to produce the output y_t and hidden state s_t for time step t.

3.2.2 Attention Mechanism

The attention mechanism [31] has been recently proposed for selecting the important parts from the input sequence for each target word. In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix. The attention mechanism has been widely used in NLP tasks. Different types of attention mechanisms have also been proposed, i.e., self-attention, multi-dimensional attention, multi-headed attention. The attention mechanism amplify the signal from the relevant part of the input sequence and provide a better representation for the input sequence.

3.3 Transformer

Ashish et al. introduced a novel architecture called Transformer [19]. Its encoder and decoder use attention mechanisms to replace the RNN. Our work applies this technique and its core part is introduced below:

- **Self-Attention** The input of Self-Attention consists of queries and keys of dimension d_k , and values of dimension d_v. We compute the attention function on a set of queries simultaneously, packed together keys(K) and values(V) into a matrix Q. We compute the matrix of outputs as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

- **Multi-Head Attention** Multi-Head Attention is a combination of multiple Self-Attention structures, each head learns features in different representation spaces, which makes the model have more capacity. Multi-Head Attention linearly projects the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively. It is computed as below:

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, ..., head_n)W^O \\ where\ head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3}$$

- **Positional Encoding** After embedding, we have a matrix representation of our tokens sequence. But these representations are not encoding the fact that tokens appear in different positions. In order for the model to make use of the order of the sequence, we need to modify the meaning represented by a specific token depending on its position. Without changing the complete representation of the token, we slightly modify it to encode its position, which is positional encoding. There are many kinds of positional encodings, learned and fixed [32]. We use sinusoidal functions described as follows – i is the position of the token in the sequence and j is the position of the embedding feature.

$$\begin{aligned} P_{i,2j} &= sin(i/10000^{2j/d_{model}}) \\ P_{i,2j+1} &= cos(i/10000^{2j/d_{model}}) \end{aligned} \tag{4}$$

- **Transformer Encoder.** The transformer encoder is composed of a stack of $N = 6$ identical layers. Each layer has a multi-head self-attention mechanism sub-layer and a position-wise fully connected feed-forward network sub-layer. There is a residual connection around each of the two sub-layers, followed by layer normalization.
- **Transformer Decoder.** The transformer decoder is also composed of a stack of $N = 6$ identical layers. Each layer has the two sub-layers that are same as encoder and a multi-head attention sub-layer. There is also a residual connection around each of the two sub-layers, followed by layer normalization.

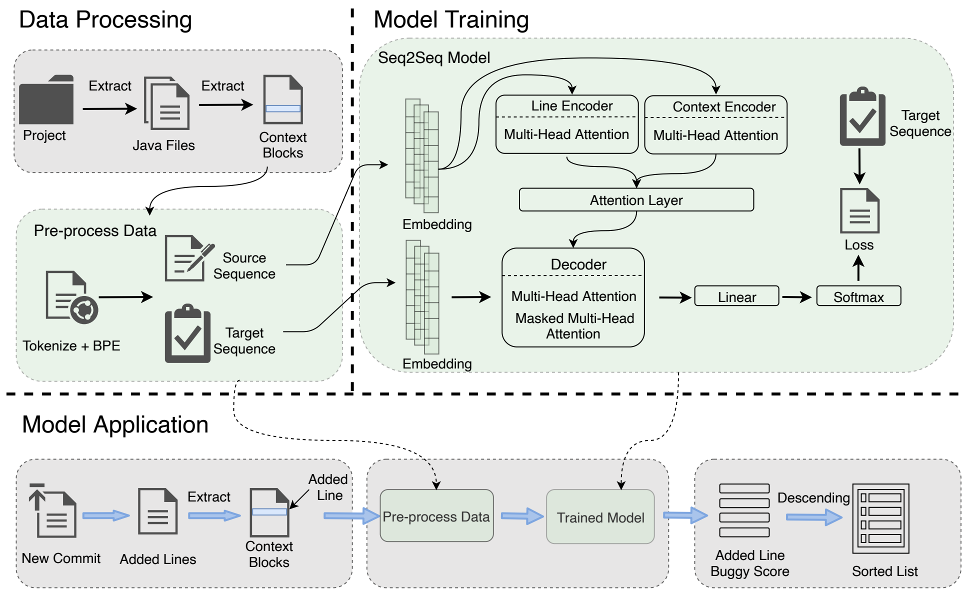


Fig. 3: Overall Framework of DEEPDL

4 APPROACH

We propose a novel deep learning based model, named DEEPDL (**Deep** Learning-based Just-in-Time **D**efect **L**ocalization), for just-in-time suspicious buggy code line location. Figure 3 outlines the details of our DEEPDL with respect to its three stages – data processing, model training and model application respectively.

4.1 Data Preparation

We use the same dataset setting by Yan et al. [12] for a fair comparison. These projects have a varying number of contributors. Besides, all the projects have over 5,000 changes to ensure sufficient samples and over 2,000 stars to ensure that the studied projects are non-trivial ones. And they have a good issue tracking system making it easy for us to label commits and source code lines. To make our paper self explanatory, we describe the details of our data preparation process as follows.

4.1.1 Collecting Training and Testing Set

In the data collecting process, we identify the clean code lines and buggy code lines respectively. Clean code lines are used as the training set for building a "Clean" neural language model. Buggy code lines are used as the testing set for evaluating the bug localization performance.

For a fair comparison, we use the same projects collected by Yan et al. [12] and choose the same settings for the *start date* and *end date* of each project. That is, for each project, we collect the changes from the start of the project to October 1, 2017. Following their experimental settings, we then identify the *splitting commit* according to the total number of changes in chronological order (60% of the commits for training and 40% of the commits for testing). The splitting commit is used to split the training and testing set. Table 1 presents the summary of the selected projects, e.g., the project name, the time period (i.e., *start date* and *end date*) we choose for each project, the total number of commits of each project during the time period.

For example, as shown in Figure 4, for the Flink project, we first count all the commits from the *start date* (*2010/12/15*) until *end date* (*2017/10/1*), which comprises 11,982 commits in total. After that, we can easily identify the *splitting commit* (happened in *2015/03/08*) for dividing the training and testing set. The data before the *splitting commit* are used for training and the data after the *splitting commit* are used for testing.

After identifying the *splitting commit*, we downloaded the snapshot of each project before the *splitting commit* for training. A snapshot represents the project's state at that point of time. To build a "Clean" neural language model, we need to make a "Clean" snapshot. In other words, we need to ensure that the downloaded snapshot only contains clean code lines. We thus need to remove all the buggy lines from the downloaded snapshot. To do this, we leverage RA-SZZ [33] to identify the clean and buggy lines both in the training set and the testing set. The detailed process is conducted as follows:

1) Bug-fix commit identification. For each commit after the splitting point, we first identify whether this commit is a *bug-fix* commit. For a given commit, if the corresponding commit message contains the defect related message (e.g., "Fixed #233"), we then check the corresponding issue report from the issue tracking system (ITS) to determine whether the report is defined as a defect. If the report is defined as a defect and it is resolved, we mark this commit as a bug-fix commit. If the report is defined as a defect and it is resolved, we then mark this commit as a *bug-fix* commit.
2) Bug-introducing commit identification. After identifying the *bug-fix* commits, for each *bug-fix* commit, we further leverage RA-SZZ [33] to identify the *bug-introducing* commits. RA-SZZ first compares the *bug-fix* commit with its previous version to identify the changed lines. Then RA-SZZ filters out the changed lines that are irrelevant to the defect changes (e.g., blank/comment lines, format modification). After that,

RA-SZZ traces back the remaining lines through the change history to identify the commits that introduce these lines, which are identified as *bug-introducing* commits.

3) Removing buggy lines from the downloaded snapshot. After identifying the *bug-introducing* commits, for each *bug-introducing* commit, if it happened before the *splitting point*, then this commit has introduced buggy lines to our training set. Therefore, we need to remove the buggy lines introduced by the *bug-introducing* commit and only retain the clean lines. Following the previous work's settings, we define the lines that were added by the *bug-introducing* commit and were later fixed by the *bug-fix* commit as buggy lines. These identified buggy lines are further mapped to the downloaded snapshot version of the project. We remove these buggy lines from the downloaded snapshot, the remaining lines can be viewed as clean code lines, which are added to the "Clean" snapshot for training a "Clean" language model.
4) Identifying the buggy lines within the testing set. For each *bug-introducing* commit, if it is happened after the *splitting point*, we add it to our testing set. Different from removing buggy code lines from our training set, we need to identify the buggy code lines in the testing set as ground truth for evaluating the localization performance. Our testing set starts from the splitting point and ends on October 1, 2017. To ensure all the buggy lines in the testing set can be correctly identified, following Yan et al's work [12], we further use a five months window (from October 1, 2017 to March 1, 2018) to cover the *bug-fix* and *bug-introducing* commits as much as possible. This is reasonable because 80% of the buggy commits are fixed within 5 months on average [12]. Then we pinpointed the buggy lines within the testing set introduced by the *bug-introducing* commits.

Finally, there are 44,244 buggy changes in the whole data set and 11,615 buggy changes for the testing set. For our training set, we first remove the buggy lines, the test code, blank lines, import statements and comments from the downloaded snapshot and retrained the rest of the source code as clean code lines, all the clean code lines are merged to make the clean snapshot for training. The summary of the training set is shown in Table 2. The clean snapshot is used for constructing code blocks to train a "Clean" neural language model. After training, the "Clean" neural language model is then used to locate the buggy lines in the testing set for evaluation. For the testing set, as shown in Table 3, we identified 11,615 *bug-introducing* commits, 1,134,601 added lines in commits and further pinpointed 109,311 buggy lines within these *bug-introducing* commits. The largest bug-introducing commit in the testing set has 2,592 added lines. The smallest bug-introducing commit in the testing set only contains 1 line of added code. The average number of added code lines within a commit is 98. The median number of added code lines within a commit is 32. And the ratio of buggy lines in a bug-introducing commit is 21.72% on average. The ratio of buggy lines in all added lines is 9.6%.

After that, we leverage the tokenize tool[1] to tokenize our training set and testing set. Our DEEPDL approach code used in our experiments and our 14 project dataset are available at https://github.com/Lifeasarain/DeepDL.

TABLE 1: Summary of Dataset

Project	Start Date	End Date	Snapshot Date	Commits
Activemq	2011/9/15	2017/9/30	2012/7/24 13:20:44 +0000	9,871
Closure-compiler	2009/11/3	2017/9/30	2016/2/10 08:21:27 -0800	10,870
Deeplearning4j	2013/11/26	2017/9/30	2016/8/31 23:12:09 +1000	8,770
Druid	2011/5/11	2017/9/30	2013/1/14 11:29:28 +0800	5,417
Flink	2010/12/15	2017/9/30	2015/3/18 10:44:43 +0100	11,982
Graylog2-server	2010/5/17	2017/9/30	2015/5/24 12:17:44 +0100	13,702
Jenkins	2006/11/5	2017/9/30	2013/5/15 18:38:49 -0400	23,764
Jetty.project	2009/3/16	2017/9/30	2014/4/7 12:52:43 +1000	14,804
Jitsi	2005/7/21	2017/9/30	2011/1/11 14:34:18 +0000	12,608
Jmeter	1998/9/3	2017/9/30	2012/2/29 13:33:18 +0000	14,625
Libgdx	2010/3/6	2017/9/30	2014/1/5 15:38:17 -0800	13,019
Robolectric	2010/7/28	2017/9/30	2014/8/21 19:21:59 -0700	7,085
Storm	2011/9/15	2017/9/30	2016/1/5 13:58:16 +0800	8,819
H2o	2014/3/3	2017/9/30	2015/9/8 13:36:41 -0700	21,914
Total				*117,250*

TABLE 2: Summary of Training Set

Project	Code Lines Before Processing	Code Blocks
Activemq	431,051	357,056
Closure-compiler	417,665	363,648
Deeplearning4j	570,009	486,080
Druid	197,770	152,431
Flink	1,011,692	855,808
Graylog2-server	199,461	174,464
Jenkins	166,077	137,280
Jetty.project	302,994	212,352
Jitsi	308,285	218,624
Jmeter	142,428	120,510
Libgdx	256,867	210,432
Robolectric	171,886	144,768
Storm	287,266	231,360
H2o	259,549	221,632
Average	*337,357*	*277,603*

TABLE 3: Summary of Test Set

Project	Commits	Buggy Commits	Total Added Lines in Buggy Commit	Buggy Lines
Activemq	3,948	597	64,156	8,433
Closure-compiler	4,348	584	38,432	3,998
Deeplearning4j	3,508	1,024	81,280	10,254
Druid	2,167	356	66,981	2,361
Flink	4,793	1,317	281,676	20,228
Graylog2-server	5,481	783	68,208	12,793
Jenkins	9,506	1,030	47,185	5,297
Jetty.project	5,922	1,089	104,832	8,322
Jitsi	5,043	1,318	115,124	13,742
Jmeter	5,850	1,265	39,796	6,536
Libgdx	5,208	609	57,612	6,144
Robolectric	2,834	418	53,132	3,818
Storm	3528	103	30,200	1104
H2o	8,766	1,122	85,987	6,281
Total	*70,902*	*11,615*	*1,134,601*	*109,311*

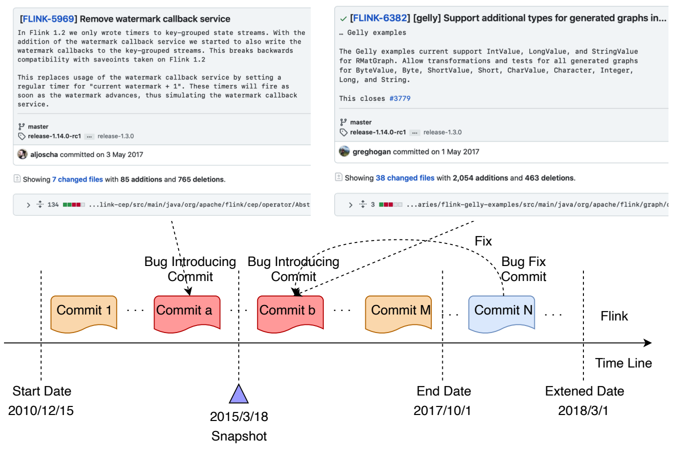
4.1.2 Processing Training and Testing Set

After collecting the training and testing set, for each project we collected, we obtained a "Clean" snapshot for training. For the "Clean" snapshot, we remove the test code, blank lines, import statements and comments. we extract the remaining source code and split the source code into a list of code lines $(l_1, l_2, ..., l_n)$. For each code line, we add the two lines preceding this line and two lines subsequent this line as its context information. That is, for a specific code line l_i, a chunk of five code lines $[l_{i-2}, l_{i-1}, l_i, l_{i+1}, l_{i+2}]$ is regarded as a basic line block L_i for building the training set.

1. https://tree-sitter.github.io

Finally, our final training set is built by stacking all the line blocks together. In summary, our training set contains a list of line blocks $[L_1, L_2, ..., L_n]$. Each line block L_i include 5 associated code lines, i.e., $L_i = [l_{i-2}, l_{i-1}, l_i, l_{i+1}, l_{i+2}]$, and each code line l_i contains a sequence of tokens.

Regarding the testing set, for each project we collected, the testing set contains a set of *bug-introducing* commits and the associated pinpointed buggy lines. Each *bug-introducing* commits contains a set of added lines (i.e., regular added lines and buggy added lines). The buggy added lines are the lines that are pinpointed as buggy for introducing bugs later. When it comes to the evaluation, since we aim to locate the buggy added lines among the added line candidates. Therefore, for each added line within the *bug-introducing* commit, we add its surrounding two lines to make a basic line block, we then feed each basic line block into our neural language model to calculate the *naturalness* score, or in other words *suspiciousness* score. The top ranked added lines are considered to be potential buggy code lines, which are used to calculate the localization performance of our model. In summary, we collected 3,886,445 code line blocks for training and 1,134,601 code line blocks for testing.



[FLINK-5969] Remove watermark callback service

```
In Flink 1.2 we only wrote timers to key-grouped state streams. With the
addition of the watermark callback service we started to also write the
watermark callbacks to the key-grouped streams. This breaks backwards
compatibility with saveoints taken on Flink 1.2

This replaces usage of the watermark callback service by setting a
regular timer for "current watermark + 1". These timers will fire as
soon as the watermark advances, thus simulating the watermark callback
service.
```

master
release-1.14.0-rc1 ... release-1.3.0
aljoscha committed on 3 May 2017

Showing 7 changed files with 85 additions and 765 deletions.

```
> 134 ...link-cep/src/main/java/org/apache/flink/cep/operator/Abst
```

✓ [FLINK-6382] [gelly] Support additional types for generated graphs in...

```
… Gelly examples

The Gelly examples current support IntValue, LongValue, and StringValue
for RMatGraph. Allow transformations and tests for all generated graphs
for ByteValue, Byte, ShortValue, Short, CharValue, Character, Integer,
Long, and String.

This closes #3779
```

master
release-1.14.0-rc1 ... release-1.3.0
greghogan committed on 1 May 2017

Showing 38 changed files with 2,054 additions and 463 deletions.

```
> 3 ...aries/flink-gelly-examples/src/main/java/org/apache/flink/graph/
```

Fig. 4: Project Time Line

4.1.3 Tokenization and Building Vocabulary

In this step, for each line block Li in the training set and testing set, we tokenize the code line into a list of tokens and then build our single vocabulary. However, due to reasons that the identifier names in the code corpus are quite arbitrary and vary greatly according to different developers, simply leveraging traditional tokenization methods on the code corpus will lead to serious Out-of-Vocabulary (OOV) problem.

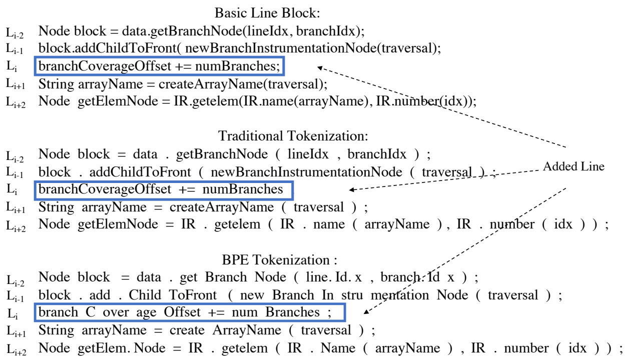
Because our model is based on neural language models, which are sensitive to the unknown tokens, too many unknown words in the testing corpus will significantly hinder the learning performance of our approach [15]. To address this challenge, Sennrich et al. [34] proposed a subword units-level model to reduce OOV problems. Following this work, Karampatsis et al. [35] applied this technique in modeling source code, which has been demonstrated to be effective in reducing OOV tokens. Inspired by their work, we first tokenize the source code line into word-level units and then we employ a Byte Pair Encoding (BPE) method [16] for subword segmentation. BPE is a data compression technique that iteratively collects the most frequent pair of bytes in a sequence and replaces it with a single unused byte. Sennrich et.al [34] first apply this technique to the word segmentation field. They merge characters or character sequences instead of bytes. They find it can actually improve performance in neural machine translation models. BPE builds up the vocabulary iteratively. For each iteration, the training corpus (in our case: a code line) is segmented into a sequence of subwords (symbols) based on the current vocabulary (a suffix symbol '@' is added to reorganize the original sequence of tokens). Following that, we count all the symbol pairs, the most frequent symbol pair (W_1, W_2) is merged and replaced with a new symbol 'W_1W_2' and added to the vocabulary. BPE algorithm takes all characters in the data set as initial vocabulary and stops after the given number of merge operations. An example of a Java code snippet tokenized into BPE subwords is shown in Figure 5.

The reasons why we adopt BPE algorithm for tokenization are as follows: (i) **The OOV problem can be alleviated**. Because our vocabulary contains more words, more unknown words in the testing set now can be represented properly. Common sequences can be represented by a single word, while the rare or unseen word will be segmented into more common subwords. (ii) **The size of the vocabulary can be significantly reduced.** Even the new identifier names proliferate as code corpus increases, we can maintain a code vocabulary with relatively small vocabulary size.

As a result, given a basic code line block $L_i = [l_{i-2}, l_{i-1}, l_i, l_{i+1}, l_{i+2}]$ from training set, for each code line $l_k (i-2 \leq k \leq i+2)$, l_k is tokenized into a sequence of subword units, i.e., $l_k = [u_{k_1}, u_{k_2}, ..., u_{k_n}]$, where $[u_{k_1}, u_{k_2}, ..., u_{k_n}]$ represents the subword unit tokens after tokenization. For example, as shown in Figure 5, the second code line "`block.addChildToFront( newBranchInstrumentationNode(traversal); `" is tokenized to a sequence of subword units ['`block`', '`.`', '`add`', '`Child`', '`ToFront`', '`new`', '`Branch`', '`In`', '`stru`', '`mentation`', '`Node`', '`(`', '`traversal`', '`)`', '`;`'] by applying the BPE tokenization method described above. When tokenizing this source code line, the BPE algorithm encounters two out of vocabulary tokens, e.g., '`addChildToFront`' and '`newBranchInstrumentationNode`'. Take '`newBranchInstrumentationNode`' as an example, BPE splits '`newBranchInstrumentationNode`' into a sequence of characters and apply the learned operations to merge the characters into larger, known word in the vocabulary. Since the words '`new`', '`Branch`', '`In`', '`stru`', '`mentation`', '`Node`' already exist in the vocabulary, so this OOV word '`newBranchInstrumentationNode`' is split into a chunk of subword unit tokens ['`new`', '`Branch`', '`In`', '`stru`', '`mentation`', '`Node`'].

Traditional tokenizers can only split the source code into tokens according to grammar, BPE can split the tokens tokenized by traditional tokenizer into finer granularity subword units. In this way, we can reduce the size of vocabulary and alleviate OOV problems.

After that, we add a special token '$\langle$`EOL`$\rangle$' to separate each line and a special token '$\langle$`EOS`$\rangle$' to the end of each basic line block. Finally, we obtained 5,021,046 code line blocks. For each code line block L_i, L_i will be tokenized as a sequence of subword units, i.e., $L_i = [l_{i-2}, \langle\texttt{EOL}\rangle, l_{i-1}, \langle\texttt{EOL}\rangle, l_i, \langle\texttt{EOL}\rangle, l_{i+1}, \langle\texttt{EOL}\rangle, l_{i+2}, \langle\texttt{EOS}\rangle]$, where $l_k = [u_{k_1}, u_{k_2}, ..., u_{k_n}]$.


Basic Line Block:

```
Li-2  Node block = data.getBranchNode(lineIdx, branchIdx);
Li-1  block.addChildToFront( newBranchInstrumentationNode(traversal);
Li    branchCoverageOffset += numBranches;
Li+1  String arrayName = createArrayName(traversal);
Li+2  Node  getElemNode = IR.getelem(IR.name(arrayName), IR.number(idx));
```

Traditional Tokenization:

```
Li-2  Node  block  =  data  .  getBranchNode  (  lineIdx  ,  branchIdx  )  ;
Li-1  block  .  addChildToFront  (  newBranchInstrumentationNode  (  traversal  )  ;
Li    branchCoverageOffset  +=  numBranches
Li+1  String  arrayName  =  createArrayName  (  traversal  )  ;
Li+2  Node  getElemNode  =  IR  .  getelem  (  IR  .  name  (  arrayName  ) ,  IR  .  number  (  idx  )  )  ;
```

BPE Tokenization :

```
Li-2  Node  block  =  data  .  get  Branch  Node  (  line. Id. x  ,  branch. Id  x  )  ;
Li-1  block  .  add  .  Child  ToFront  (  new  Branch  In  stru  mentation  Node  (  traversal  )  ;
Li    branch  C  over  age  Offset  +=  num  Branches  ;
Li+1  String  arrayName  =  create  ArrayName  (  traversal  )  ;
Li+2  Node  getElem. Node  =  IR  .  getelem  (  IR  .  Name  (  arrayName  )  ,  IR  .  number  (  idx  )  )  ;
```

Fig. 5: Tokenize example

4.2 Model Training

4.2.1 Model Overview

Naturalness represents how "surprised" an element is by a given document. Previous studies have demonstrated the effectiveness of using a language model to capture the "*naturalness*" of software [14]. They found that buggy code is rated as significantly more "unnatural" by language models. Inspired by these findings, we propose a neural language model, DeepDL, to locate the suspicious code elements when the code change happens.

After the data preparation process, all the training line blocks are tokenized into subword unit sequences. For a given processed line block $L_i = [l_{i-2}, \langle\texttt{EOL}\rangle, l_{i-1}, \langle\texttt{EOL}\rangle, l_i, \langle\texttt{EOL}\rangle, l_{i+1}, \langle\texttt{EOL}\rangle, l_{i+2}, \langle\texttt{EOS}\rangle]$, we want to build a neural language model to estimate the naturalness of the central source code line (i.e., l_i) with respect to its context (i.e., $[l_{i-2}, l_{i-1}, l_{i+1}, l_{i+2}]$). By this we mean for a given code block, whether the central code line is "natural" with respect to their surrounding lines. We formulated this task as a sequence-to-sequence (Seq2Seq) learning problem, which turns one sequence (i.e., the source sequence) into another sequence (i.e., the target sequence). The primary components of the Seq2Seq model are encoder and decoder network. The encoder turns each item within the source sequence into a corresponding hidden vector, while the decoder reverses the process, turning the vector into a target sequence item.

In this study, we set the source sequence X_{src} as the whole line block L_i (i.e., $X_{src} = L_i$), and we set the target sequence Y_{tgt} as central code line of each code block (i.e., $Y_{tgt} = [l_i]$). Mathematically, given X_{src} is a sequence of tokens within the code line block, our neural language model aims to generate the central code line Y_{tgt}, which is "natural" to its context. Overall, our goal is to train a language model θ using $\langle X_{src}, Y_{tgt}\rangle$ pairs, such that the probability $P_\theta(\mathbf{Y_{tgt}}|\mathbf{X_{src}})$ is maximized over the given training dataset, $P_\theta(\mathbf{Y_{tgt}}|\mathbf{X_{src}})$ can be seen as the conditional log-likelihood of the central code line given the code block input.

4.2.2 Encoders

Our model follows a sequence-to-sequence architecture and the encoder part learns latent features from source code lines. Recently, Transformers have been widely used to capture the code semantic features by encoding code into vectors [36]. In this study, we employ a Transformer Encoder [19] as the encoder template for our task. The transformer encoder is composed of a stack of 6 residual encoder blocks, each encoder block is broken down into two sub-layers (i.e., a self-attention sub-layer and a feed-forward network sub-layer). The input to the transformer encoder is a sequence of tokens, the input sequence of tokens flows through each of the two layers of the encoder components. The first encoder block transforms the input sequence from a context-independent token representation to a context-dependent vector representation, and the following encoder blocks further refine this contextual representation until the last encoder block outputs the final contextual encoding. The output of the transformer is a contextualized vector of the input sequence.

To better estimate the "naturalness" of a code line block, we adopt two encoders, i.e., *Central Line Encoder* and *Context Line Encoder* to embed the central line and context lines into vector representation respectively. In this study, *Central Line Encoder* and *Context Line Encoder* are the same in structure which use the transformer encoder. Likewise, the input to the encoders is a basic code line block L_i, the outputs of the encoders are two embedding vectors respectively. Through the two encoders, the semantically related concepts across different source code lines can be mapped and correlated in the higher dimensional vector space.

- *Central Line Encoder.* For a basic code line block L_i, the *Central Line Encoder* extracts out the central line (i.e., l_i) and uses the transformer encoder to embed it into a semantic vector x_{cen}.
- *Contextual Line Encoder.* For a basic code line block L_i, we extract out the two lines before the central line (i.e., $[l_{i-2}, l_{i-1}]$) as the preceding context, and two lines after the central line (i.e., $[l_{i+1}, l_{i+1}]$) as the subsequent context. Similar to the *Central Line Encoder*, After feeding the preceding context and the subsequent context into the *Context Line Encoder*, we can get the embedding vector x_{con} for the context lines within a code line block.

4.2.3 Decoders

The decoder's job is to generate the target sequence. Similar with the transformer encoder, the transformer decoder has similar sub layers. The transformer decoder is composed of 6 decoder blocks. Each decoder block has two self-attention layers, and a feed-forward layer. The decoder is capped off with a linear layer and a softmax layer to get the final word probability distributions. The decoder is auto-regressive, in particular, it takes the encoder's contextualized vectors as well as the previous outputs as inputs, and generates a single output step by step.

To connect the Encoder and Decoder, we employ a cross-attention layer. In particular, after getting the central line vector x_{cen} and the context line vector x_{con}, the cross-attention layer takes x_{cen} and x_{con} as input and outputs a hidden state vector x_h. We then send x_h into our transformer decoder, the transformer decoder will turn the hidden vector into a target sequence. Mathematically, given the hidden states x_h, the transformer decoder calculates the conditional probability distribution of the target sequence Y_{tgt}, i.e., $P_\theta(\mathbf{Y_{tgt}}|\mathbf{x_h})$, as follows:

$$P_\theta(\mathbf{Y_{tgt}}|\mathbf{x_h}) = \prod_{i=1}^{L} P_\theta(\mathbf{y_i}|\mathbf{Y_{0:i-1}};\mathbf{x_h}) \tag{5}$$

where L is the length of the target sequence Y_{tgt}. The transformer decoder first maps the encoded hidden states (i.e., x_h) and all the previous target states $Y_{0:i-1}$, to logit vector l_i. The logit vector l_i is then processed by the softmax operation to estimate the conditional distribution $P_\theta(\mathbf{y}_i|\mathbf{Y}_{0:i-1};\mathbf{x}_h)$. After calculating the above conditional distribution, we can auto-regressively generate the output sequence and thus define a mapping of an input sequence X_{src} to an output sequence Y_{tgt}.

4.2.4 Data Flow

We summarize the data-flow of our model as follows: as shown in Figure 6, the input to our model is a basic code line block L_i, which is broken into two parts (the central code line $[l_i]$ and the context code lines $[l_{i-2}, l_{i-1}, l_{i+1}, l_{i+2}]$). Each code line is represented as a sequencee of subword unit tokens. Then the central code line is passed through the *Central Line Encoder* to generate the central line encoded vector x_{cen}, while the context code lines are passed through the *Context Line Encoder* to generate the context lines encoded vector x_{con}. After that, a cross attention layer takes the x_{cen} and x_{con} as input and outputs a hidden state vector x_h, which can capture the relationship between the central code line and the context code lines. The hidden state vector x_h is then passed through the Decoder part to generate the target sequence. The Decoder part takes in the encoded hidden states (i.e., x_h) and step by step generates a single output y_i while also being fed the previous output $Y_{0:i-1}$. To be more specifically, the transformer decoder first maps the hidden state vector (i.e., x_h) as well as the previous output $Y_{0:i-1}$ to a logit vector l_i, the logit vector l_i then goes through a final softmax layer to model the conditional probability distribution of the target sequence. The softmax layer will produce a probability distribution vector over all vocabulary tokens, and we choose the token with the highest probability as the predicted token.

4.2.5 Loss Function

We leverage a cross entropy loss function to calculate the loss of the model. The cross entropy (entropy in short) is a widely-adopted metric used in statistical language models, a sentence with higher entropy score is considered to be more natural. Ray et al. [14] investigated the possibility of using entropy to estimate the "naturalness of buggy code". The core research question of their work is "can entropy provide a useful indication of the likely bugginess of a line of code?". According to their experimental results, they found that buggy code lines have higher entropy scores than non-buggy lines, which means the entropy can be an indicator to measure the naturalness (or suspiciousness) of a code snippet. The higher entropy of a code snippet, the

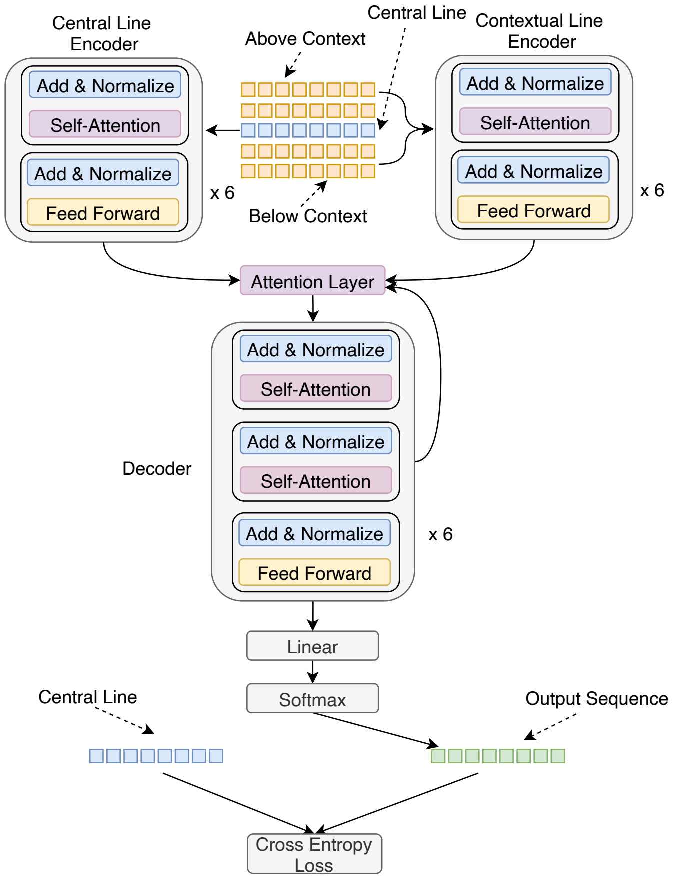


Fig. 6: Architecture of Proposed Approach

more unnatural (or suspicious) the code snippet is with the training corpus. In particular, regarding the decoding process, the probability of generating a token y_i is $P(y_i)$. During the training process, for each token at each timestamp, the loss associated with the generated central code line is $-\frac{1}{l}\sum_{i=1}^{l}\log_2 p(y_i)$, where l is the length of the central code line. The final goal of our model is to minimize the cross entropy, i.e., minimize the following objective function over all the training dataset:

$$H(y) = -\frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{l}\log_2 p(y_i^{(j)}) \tag{6}$$

where N is the number of training instances, $y_i^{(j)}$ represents the ith token in the jth training sample. The cross entropy describes how much the predicted probability diverges from the ground truth. Through optimizing the above objective loss function using optimization algorithms (e.g., gradient descendant), the parameters θ of our model can be estimated. Finally, after the training process, we can obtain a neural language model (i.e., DEEPDL). The neural language model maximizes the probability of the target sequence given the input sequence (i.e., $P_\theta(\mathbf{Y_{tgt}}|\mathbf{X_{src}})$) over our training dataset.

4.3 Model Application

For practical application, the input of DEEPDL is a buggy commit (identified by the JIT defect prediction tools) or a newly submitted commit. Given a buggy code commit, DEEPDL first extracts all the added code lines within this commit. For each added line, we make a code line block by adding its surrounding two lines, we process the code line block as described in Section 4.1.1 and Section 4.1.2. For each code line block, we fed the code line block into our trained neural language model, DEEPDL, to generate an output sequence. The generated sequence can be considered as a "clean" code line since it is generated from our trained "clean" neural language model. After that, we can calculate the entropy between the generated sequence and the added line, and the entropy of the added code line can be computed as the average of the entropy of each subword token within this code line, as follows:

$$H_p(s) = \frac{1}{|s|}\sum_{n=1}^{|s|} H_p(t_i) \tag{7}$$

Finally, we get the entropy of all the added lines in the buggy commits and treat the entropy as its suspiciousness score. The code line with the highest suspiciousness score is considered to be a possible defect location in the code. Similar to the buggy commit, if we are handling the newly submitted commits, by following the same application pipeline, DEEPDL can identify the suspicious added lines within the newly submitted commits, which can reduce the risk of introducing bugs and improve the software's reliability.

5 EXPERIMENT SETUP

We first introduce our data preparation process, then present the detailed parameter settings for training our **DEEPDL** approach. We then introduce our chosen evaluation metrics used in this study for evaluating the performance of our approach.

5.1 Training Details

We set the initial learning rate to 0.1 with a momentum of 0.5 and clip the gradients norm by 5. The learning rate decay of 0.99. The size of mini-batches is 16. Our model is trained using the Stochastic Gradient Descent (SGD) algorithm. We use the cross-entropy as the loss function. It is worth mentioning that for each project, we reserve 10% of the training set as the validation set. We further tuned the hyperparameters according to the performance of the model on the validation set. Specifically, for each project, the training runs for 50 epochs and we save the model after each epoch, we then select the model with the best performance (the lower of the entropy score, the better performance of the neural language model) on validation set as our final neural language model. We build our model based on Pytorch [2] using four NVIDIA RTX 2080Ti GPU.

5.2 Evaluation Measure

To evaluate the performance of our approach, we use the widely accepted metrics MRR (Mean Reciprocal Rank), MAP (Mean Average Precision) [37] and Top-k Accuracy as the evaluation metrics. In addition, these evaluation metrics are also adopted in Yan et al's work. Thus they can be used for fair comparison purposes We introduce the details of these three evaluation metrics as follows.

2. https://pytorch.org

5.2.1 MRR

MRR is a popular metric used to evaluate an information retrieval technique [37]. For a given query, its reciprocal rank is the multiplicative inverse of the rank of the first correct answer. For our study, MRR measures how far we need to check down a sorted list of added lines of a buggy change to locate the first buggy line. It can be computed as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{8}$$

where Q is the number of queries. MRR is the average of the reciprocal ranks for queries Q.

5.2.2 MAP

MAP provides the mean of the average precision scores for a set of queries. The average precision (AP) of a query is the average of the precision values for this query. MAP considers the ranks of all buggy lines in that sorted list. It can be computed as:

$$AP = \sum_{i=1}^{M}\frac{P(i) * rel(i)}{numberofrelevantdocuments} \tag{9}$$

where i is then rank in the sequence of retrieved item, $P(i)$is the precision at cut-off i in the list. $rel(k)$ is a indicator function equaling 1if the item at rank i is a relevant item

$$MAP = \frac{1}{N}\sum_{i=1}^{N} AP_i \tag{10}$$

Our evaluation is performed at the change-level. Each buggy change in our test set has a MRR and a MAP performance value. The higher MRR and MAP value means that the model has a better bug localization performance.

5.2.3 Top-K Accuracy

Top-k Accuracy measures whether the Top-k most likely buggy lines returned by our approach are actually the buggy code location. For example, given one defect change c, if at least one of the Top-k most likely buggy lines returned by our approach is actually the buggy location, we regard the localization as successful, and set the Top-k value of this change $Topk(c)$ to 1; otherwise, we regard the localization as unsuccessful and set the Top-k value $Topk(c)$ to 0. Consider a set of N defect changes in a project P, its Top-k accuracy is computed as:

$$Topk(P) = \frac{1}{N}\sum_{c=1}^{N} Topk(c). \tag{11}$$

Following the experimental settings in previous studies [12], in this paper, we set $k = 1$ and 5.

TABLE 4: The performance DEEPDL vs. Baselines

Project	Top-1 accuracy			Top-5 accuracy			MRR			MAP		
	Yan's	CC2Vec	Ours	Yan's	CC2Vec	Ours	Yan's	CC2Vec	Ours	Yan's	CC2Vec	Ours
Activemq	0.3936	0.3417	**0.4070**	0.5628	0.5142	**0.6348**	0.4785	0.4317	**0.5117**	0.4505	0.4217	**0.4749**
Closure-compiler	0.2432	0.2226	**0.2774**	0.4966	0.4452	**0.4983**	0.3650	0.3404	**0.3933**	0.3504	0.3289	**0.3718**
Deeplearning4j	0.2451	0.1963	**0.3184**	0.4971	0.4189	**0.5908**	0.3702	0.3150	**0.4464**	0.3264	0.2847	**0.3746**
Druid	**0.2107**	0.1517	0.1966	0.4157	0.2949	**0.4663**	0.3149	0.2330	**0.3297**	0.2720	0.2043	**0.2975**
Flink	0.2065	0.1716	**0.2445**	0.4146	0.3470	**0.4761**	0.3125	0.2625	**0.3584**	0.2608	0.2307	**0.2894**
Graylog2-server	0.3384	0.3207	**0.3742**	0.5951	0.5249	**0.6731**	0.4637	0.4207	**0.5077**	0.4222	0.3961	**0.4503**
Jenkins	0.3602	0.2951	**0.3748**	0.6184	0.5436	**0.6699**	0.4834	0.4149	**0.5149**	0.4536	0.4026	**0.4792**
Jetty	0.2452	0.1928	**0.2773**	0.4702	0.4279	**0.5427**	0.3550	0.3104	**0.4022**	0.3210	0.2838	**0.3537**
Jitsi	0.3475	0.3126	**0.3634**	0.6297	0.5849	**0.6798**	0.4798	0.4373	**0.5043**	0.4325	0.4007	**0.4425**
Jmeter	0.4545	0.4103	**0.4980**	0.7462	0.6743	**0.7968**	0.5838	0.5336	**0.6285**	0.5615	0.5188	**0.6048**
Libgdx	0.3448	0.2808	**0.3711**	0.6010	0.5107	**0.6568**	0.4668	0.3945	**0.4981**	0.4258	0.3740	**0.4582**
Robolectric	0.2368	0.1699	**0.2536**	0.4928	0.4115	**0.5383**	0.3654	0.2920	**0.3851**	0.3145	0.2662	**0.3301**
Storm	0.0971	0.0874	**0.1748**	0.3495	0.2524	**0.3689**	0.2062	0.1840	**0.2789**	0.1810	**0.2338**	**0.2338**
H2o	0.2584	0.2299	**0.3057**	0.5258	0.4483	**0.5954**	0.3854	0.3427	**0.4427**	0.3542	0.3197	**0.3996**
average	*0.2844*	*0.2416*	***0.3169***	*0.5297*	*0.4571*	***0.5849***	0.4022	*0.3509*	***0.4430***	*0.3662*	*0.3197*	***0.3972***
p-value	*<0.001*	*<0.001*		*<0.001*	*<0.001*		*<0.001*	*<0.001*		*<0.001*	*<0.001*	

TABLE 5: The performance Cross-Project model vs. Within-Project model

Project	Top-1 accuracy			Top-5 accuracy			MRR			MAP		
	CP	WP	Improve	CP	WP	Improve	CP	WP	Improve	CP	WP	Improve
Activemq	**0.4154**	0.4070	-2.01%	0.6181	**0.6348**	2.71%	**0.5146**	0.5117	-0.57%	**0.4779**	0.4749	-0.63%
Closure-compiler	0.2759	**0.2774**	0.54%	**0.5017**	0.4983	-0.68%	**0.3962**	0.3933	-0.73%	**0.3743**	0.3718	-0.66%
Deeplearning4j	0.3174	**0.3184**	0.30%	0.5879	**0.5908**	0.50%	0.4442	**0.4464**	0.50%	0.3720	**0.3746**	0.71%
Druid	**0.2022**	0.1966	-2.76%	0.4579	**0.4663**	1.83%	0.3249	**0.3297**	1.48%	0.2896	**0.2975**	2.74%
Flink	**0.2498**	0.2445	-2.12%	**0.4951**	0.4761	-3.84%	**0.3640**	0.3584	-1.54%	**0.2900**	0.2894	-0.21%
Graylog2-server	**0.3908**	0.3742	-4.25%	**0.6731**	0.6731	-0.01%	**0.5169**	0.5077	-1.78%	**0.4536**	0.4503	-0.73%
Jenkins	0.3592	**0.3748**	4.33%	0.6573	**0.6699**	1.92%	0.5002	**0.5149**	2.94%	0.4670	**0.4792**	2.62%
Jetty	0.2617	**0.2773**	5.97%	0.5298	**0.5427**	2.43%	0.3870	**0.4022**	3.93%	0.3434	**0.3537**	3.00%
Jitsi	**0.3695**	0.3634	-1.64%	0.6737	**0.6798**	0.91%	**0.5068**	0.5043	-0.50%	**0.4428**	0.4425	-0.08%
Jmeter	0.4830	**0.4980**	3.11%	0.7794	**0.7968**	2.24%	0.6146	**0.6285**	2.26%	0.5946	**0.6048**	1.71%
Libgdx	**0.3744**	0.3711	-0.88%	0.6355	**0.6568**	3.35%	**0.4982**	0.4981	-0.02%	0.4565	**0.4582**	0.38%
Robolectric	0.2512	**0.2536**	0.95%	**0.5526**	0.5383	-2.59%	**0.3856**	0.3851	-0.13%	**0.3361**	0.3301	-1.77%
Storm	0.1650	**0.1748**	5.91%	**0.4078**	0.3689	-9.53%	0.2758	**0.2789**	1.11%	**0.2341**	0.2338	-0.13%
H2o	**0.3128**	0.3057	-2.27%	0.5865	**0.5954**	1.52%	**0.4453**	0.4427	-0.58%	0.3975	**0.3996**	0.53%
average	*0.3163*	***0.3169***	*0.19%*	*0.5826*	***0.5849***	*0.39%*	*0.4410*	***0.4430***	*0.45%*	*0.3950*	***0.3972***	*0.56%*
p-value	*>0.05*			*<0.05*			*<0.001*			*<0.001*		

6 EMPIRICAL EVALUATION

We evaluate the performance of our new DEEPDL approach on 14 open source projects. We attempt to answer the following key research questions:

- RQ1: How effective is our approach compared with the state-of-the-art baselines?
- RQ2: How effective is our approach when using cross-project modeling?
- RQ3: How effective is our approach for using of context information and BPE tokenization methods?

6.1 RQ1: Effectiveness Evaluation

6.1.1 Experimental Setup.

To evaluate the effectiveness of our model, we conducted extensive experiments on the selected 14 projects. We use our trained neural language model, DEEPDL, to predict the locations of buggy lines within the test set. We compare DEEPDL with the following state-of-the-art models for comparison purposes:

- Yan's approach. Yan's approach is currently the state-of-the-art JIT defect localization approach. It estimates software naturalness with the N-gram language model, which can locate suspicious defective lines in a defect change at check-in time. Different from building the n-gram language model, in this study, we employ the transformer based encoder and decoder to make an neural language model.
- CC2Vec. CC2Vec is the state-of-the-art defect prediction tool. CC2Vec is an embedding-based approach proposed by Hoang et al. [17]. Different from the JIT defect localization task, CC2Vec is designed for the JIT defect prediction task. For a given commit, CC2Vec learns two embedding vectors from the log message and code change and outputs a probability to judge if this commit is buggy or not. To adapt this JIT defect prediction tool to our task of JIT defect localization, for a given commit, we regard each added line of this commit as a single commit, then the added line is passed through CC2Vec to produce a probability indicating that this added line is buggy. The added lines with highest probability scores will be considered as potential buggy lines for this commit. It is worth to mention that for a fair comparison, we drop the log message related features and only keep the code change part for CC2Vec, this is reasonable because DEEPDL only model the source code without considering additional information.

6.1.2 Experimental Results.

Table 4 illustrates the Top-1 and Top-5 accuracy, MRR, MAP of our approach and the baselines. We can observe the following points from the table:

1) **The CC2Vec model achieves the worst performance regarding different evaluation measurements.** The CC2Vec model is originally designed for the task of JIT defect prediction. It formulates the JIT defect prediction task as a binary classification problem, that is given a commit, CC2Vec outputs a probability score to judge this commit is buggy or not. We transfer their approach from predicting a buggy commit to predicting a buggy line (treating each separate line as a commit). The suboptimal performance of CC2Vec indicates that the binary classification strategy is not suitable for the task of JIT defect localization. This is because CC2Vec treats a single added line as input, it is thus unable to consider the preference relationship among different code lines. By contrast, the language model based approaches (including Yan's approach and ours) models the historical clean code lines, which can estimate the naturalness of the source code.
2) Our approach outperforms Yan's approach in terms of all measures on average. From the table 4, we can see that our approach can achieve a higher accuracy than Yan's approach with the defect localization task. For example, the improvement of our approach over Yan's approach is 11.42% for Top-1 accuracy and 9.69% for Top-5 accuracy, while 11.35% for MAP and 9.55% for MRR scores. We attribute this to the following reasons: First, both DEEPDL and Yan's approach adopt the language model, however, Yan's approach builds the language model with n-grams, which can only capture the lexical level features. In this study, we use the transformer based encoder and decoder to construct a neural language model, which not only considers the lexical level features but also semantic level features. Second, we adopt the BPE algorithm for tokenization, which can solve the OOV problem when dealing with the testing set.
3) **Regarding all the 14 projects, the improvements of our proposed model over baseline are significant.** To test the statistical significance, we employ the Wilcoxon signed-rank test [38] with a Bonferroni correction [39] at 95% confidence level. The Wilcoxon signed-rank test is a non-parametric hypothesis test that used to compare two matched samples to assess whether their population mean ranks differ, while Bonferroni correction is used to counteract the problem of multiple comparisons. From the table, we can see that on average all the p-values are substantially smaller than 0.05, which shows that the improvements of our proposed model are statistically significant.
4) Only for the project "Druid" is the Top-1 accuracy of our approach is lower than the baseline. This is because the number of bug introducing changes in this project is small and the bug introducing lines are relatively low in the added lines in most bug introducing changes. Both our approach and the baseline do not perform well on the top-1 accuracy. Except for this indicator, Our approach outperforms the baseline in terms of all the measures, we argue that the improvement of our approach is significant.

Answer to RQ1: How effective is our approach compared with the state-of-the-art baseline? – we conclude that our approach significantly outperforms the baseline and achieves a new state-of-the-art performance for just-in-time defect localization.

6.2 RQ2: Cross-Project Evaluation

6.2.1 Experimental Setup

A cross-project defect localization technique trains the localization model by using data from other source projects and uses the trained model to perform defect localization for the target project. To measure the performance of our approach in cross-project defect localization, we build our DEEPDL model by learning from all other projects. To identify defects in the target project, it follows a two-step process, model building step and model application step. In the model building step, we first combine all the training data of other projects except the target one as a multi-project training set. A specific localization model is then built based on this corpus using the same setting in RQ1. During the model application step, we choose the target project as testing set and run the model on this set. Finally, we compare the performance of the cross-project model with the within-project model.

6.2.2 Experimental Results

Table 5 shows the performance of the cross-project model and corresponding within-project model. From the table, we can see that **the cross-project model achieves a comparable performance to the within-project model.** In all projects, the within-project model achieves a slightly better performance on average compared with the cross-project model. For example, the average Top-1, Top-5, MRR and MAP score of cross-project model are 0.3163, 0.5826, 0.441, 0.395, while the within-project model achieves very close performance of 0.3169, 0.5849, 0.4430 and 0.3972 respectively.

We can conclude that a cross-project defect prediction model is feasible. This is because the training corpus in the cross-project is much larger than the corpus used for the within-project. For example, in our cross-project setting, the size of training corpus is about 236MB on average. In our within-project setting, the size of training corpus is about 18MB on average. The cross-project training corpus is 13 times larger than the within-project training corpus. When our DEEPDL model is trained with the larger cross-project data, it successfully capture the program semantics and automatically learns the naturalness of the code from different types of projects, this also justifies the robustness and generalize ability of our model. This enlightens us that we can train a defect localization model with a large training corpus, and then apply it to new projects in future applications. While the model has achieved good performance, we have saved a lot of training time and the model is more versatile.

Answer to RQ2: How effective is our approach when using cross-project modeling? – we conclude that training our approach with cross-project data is feasible and can achieve comparable performance as the within-project setting.

TABLE 6: The performance without BPE model vs. enhanced model

Project	Top-1 accuracy			Top-5 accuracy			MRR			MAP		
	WB	Original	Improve	WB	Original	Improve	WB	Original	Improve	WB	Original	Improve
Activemq	0.3735	**0.4070**	8.97%	0.5796	**0.6348**	9.54%	0.4762	**0.5117**	7.44%	0.4580	**0.4749**	3.68%
Closure-compiler	**0.2894**	0.2774	-4.14%	**0.5086**	0.4983	-2.02%	**0.3960**	0.3933	-0.67%	0.3701	**0.3718**	0.47%
Deeplearning4j	0.2988	**0.3184**	6.54%	0.5518	**0.5908**	7.08%	0.4215	**0.4464**	5.91%	0.3546	**0.3746**	5.65%
Druid	**0.2360**	0.1966	-16.67%	0.4522	**0.4663**	3.11%	**0.3449**	0.3297	-4.42%	0.2968	**0.2975**	0.26%
Flink	0.2103	**0.2445**	16.25%	0.4366	**0.4761**	9.04%	0.3250	**0.3584**	10.28%	0.2668	**0.2894**	8.47%
Graylog2-server	0.3729	**0.3742**	0.34%	0.6054	**0.6731**	11.18%	0.4843	**0.5077**	4.84%	0.4314	**0.4503**	4.38%
Jenkins	0.3670	**0.3748**	2.12%	0.6379	**0.6699**	5.02%	0.4950	**0.5149**	4.02%	0.4517	**0.4792**	6.09%
Jetty	0.2525	**0.2773**	9.82%	0.4995	**0.5427**	8.64%	0.3756	**0.4022**	7.09%	0.3272	**0.3537**	8.10%
Jitsi	0.3422	**0.3634**	6.21%	0.6351	**0.6798**	7.05%	0.4798	**0.5043**	5.09%	0.4211	**0.4425**	5.07%
Jmeter	0.4957	**0.4980**	0.48%	0.7897	**0.7968**	0.90%	0.6285	**0.6285**	0.00%	0.5992	**0.6048**	0.93%
Libgdx	0.3415	**0.3711**	8.65%	0.6190	**0.6568**	6.10%	0.4647	**0.4981**	7.19%	0.4238	**0.4582**	8.13%
Robolectric	0.2297	**0.2536**	10.42%	0.4785	**0.5383**	12.50%	0.3529	**0.3851**	9.13%	0.3089	**0.3301**	6.87%
Storm	0.1650	**0.1748**	5.88%	**0.3883**	0.3689	-5.00%	0.2726	**0.2789**	2.30%	0.2137	**0.2338**	9.42%
H2o	0.2941	**0.3057**	3.94%	0.5793	**0.5954**	2.78%	0.4306	**0.4427**	2.81%	0.3814	**0.3996**	4.77%
average	*0.3049*	***0.3169***	*3.94%*	*0.5544*	***0.5849***	*5.50%*	*0.4248*	***0.4430***	*4.27%*	*0.3789*	***0.3972***	*4.82%*
p-value	*<0.001*			*<0.001*			*<0.001*			*<0.001*		

TABLE 7: The performance Without context learning model vs. enhanced model

Project	Top-1 accuracy			Top-5 accuracy			MRR			MAP		
	WC	Original	Improve	WC	Original	Improve	WC	Original	Improve	WC	Original	Improve
Activemq	0.3819	**0.4070**	6.58%	0.6147	**0.6348**	3.27%	0.4930	**0.5117**	3.79%	0.4612	**0.4749**	2.96%
Closure-compiler	0.2723	**0.2774**	1.89%	**0.5188**	0.4983	-3.96%	0.3932	**0.3933**	0.04%	0.3687	**0.3718**	0.85%
Deeplearning4j	0.2734	**0.3184**	16.43%	0.5566	**0.5908**	6.14%	0.4100	**0.4464**	8.88%	0.3588	**0.3746**	4.41%
Druid	0.1798	**0.1966**	9.38%	0.4466	**0.4663**	4.40%	0.3101	**0.3297**	6.32%	0.2837	**0.2975**	4.89%
Flink	0.2422	**0.2445**	0.94%	0.4655	**0.4761**	2.28%	0.3515	**0.3584**	1.95%	0.2866	**0.2894**	0.98%
Graylog2-server	0.3678	**0.3742**	1.74%	**0.6807**	0.6731	-1.13%	0.5058	**0.5077**	0.37%	**0.4541**	0.4503	-0.84%
Jenkins	0.3699	**0.3748**	1.31%	0.6563	**0.6699**	2.07%	0.5064	**0.5149**	1.67%	0.4714	**0.4792**	1.66%
Jetty	0.2498	**0.2773**	11.03%	0.5271	**0.5427**	2.96%	0.3797	**0.4022**	5.94%	0.3364	**0.3537**	5.12%
Jitsi	**0.3680**	0.3634	-1.24%	0.6715	**0.6798**	1.24%	**0.5073**	0.5043	-0.60%	**0.4451**	0.4425	-0.60%
Jmeter	0.4964	**0.4980**	0.32%	0.7834	**0.7968**	1.72%	0.6241	**0.6285**	0.71%	0.5986	**0.6048**	1.03%
Libgdx	0.3662	**0.3711**	1.35%	0.6289	**0.6568**	4.44%	0.4898	**0.4981**	1.70%	0.4487	**0.4582**	2.12%
Robolectric	0.2368	**0.2536**	7.07%	0.5144	**0.5383**	4.65%	0.3693	**0.3851**	4.27%	0.3247	**0.3301**	1.66%
Storm	**0.1845**	0.1748	-5.26%	**0.4466**	0.3689	-17.39%	**0.2988**	0.2789	-6.68%	**0.2382**	0.2338	-1.84%
H2o	0.2825	**0.3057**	8.20%	0.5517	**0.5954**	7.92%	0.4143	**0.4427**	6.86%	0.3770	**0.3996**	5.99%
average	*0.3068*	***0.3169***	*3.28%*	*0.5778*	***0.5849***	*1.23%*	*0.4338*	***0.4430***	*2.13%*	*0.3905*	***0.3972***	*1.71%*
p-value	*<0.001*			*<0.001*			*<0.001*			*<0.001*		

6.3 RQ3: Ablation Evaluation

6.3.1 Experimental Setup

Our approach adds two enhancements to the original Seq2Seq model: using BPE method in the tokenization step for solving the OOV problems, and using the code line's context for better representing the program semantics. To evaluate the performance of our approach of incorporating these two techniques, we also perform an ablation analysis to investigate if such enhancements significantly improve the performance of our approach. To do this we compare the performance of DEEPDL with its two variants as follows:

- **WB (Without BP) Model**: WB model drops the BPE tokenization technique in data processing stage, and replaces it with the traditional tokenization method.
- **WC (Without Context) Model**: WC model drops the context information we added to the code lines, and trains the DEEPDL with a single line instead.

6.3.2 Experimental Results

Table 6 and Table 7 demonstrates the performance of our approach compared with **WB** and **WC** respectively. From the tables, we can see that **the DEEPDL outperforms the WB model and the WC model by a large margin on average.** Regarding the Top-1 accuracy, The improvement of our approach over **WB** is 3.94% and the improvement over **WC** is 3.28% on average. The evaluation result verifies the importance and necessity of these two techniques incorporated within DEEPDL, and further confirms their usefulness for enhancing the performance of defect localization.

By dropping the BPE tokenization method from data processing, the average number of unknown words sharply increases from 68 to 6215 (about 100 times larger). When there are too many OOV tokens in the testing set, the naturalness score estimated by DEEPDL will be greatly affected by these unknown words. It can no longer effectively calculate the likelihood of a buggy code line. Under such conditions, the bug probability score we calculated is also inaccurate and unreliable.

By dropping the context of the code line and treating each single code line as input, the model loses much valuable information of this code line. However, the adjacent code lines are often closely connected with each other and should be considered as an united block. The performance drop between our approach and WC further justifies our assumption, that the context information do have contributions on the overall performance of our approach.

Answer to RQ3: How effective is our approach for using of context information and BPE tokenization methods? – we conclude that both the context information and BPE method are effective and helpful to enhance the performance of our approach.

6.4 Threats to Validity

Threats to internal validity refer to errors in our experiments. For each task, we carefully reuse existing implementations of the baseline approach [12]. We have double checked our code and implementation, but errors may remain.

Threats to external validity concern the generalizability of our approach. In our experiment, we only consider Java software projects. Although the results on Java have proved the effectiveness of our approach, we do not verify the generality of our approach to projects written in other programming languages, which may be a threat to external validity. When doing further research in other programming languages, some steps should be carefully adapted. For example, removing testing files, removing comments and code tokenization should follow different programming language rules. In the future, we will extend our approach to other programming languages to mitigate this threat.

Threats to the quality of collected dataset. We collected the bug introducing changes and bug introducing code lines from the open source projects as other work does. Although the mining method RA-SZZ [33] can deal with noise including blank/comment lines, format modifications and refactoring modifications, there is some noise in our dataset. In the future, we will investigate a better technique to build a better dataset.

Threats to the model validity relates to model structure that could affect the learning performance of our approach. In this study, we choose the transformer-based encoder and decoder to build a neural language model for capturing the naturalness of the source code. Recent research has proposed new models, such as BERT [40], ALBERT [41], GPT [42], that can achieve better performance than transformer. However, our results do not shed light on the effectiveness of employing other advanced models with respect to different structures and new features. We will try to use other deep learning models for our tasks in future work.

7 Discussion

7.1 Impact of Different Methods for Using Entropy

When submitting code changes, in order to detect the localization of the possible buggy line, our tool DeepDL sorts all of the added lines according to their line entropy. Therefore, the way we calculate the line entropy is an important factor to affect the performance of our approach. For a given line, it is intuitive to represent the line entropy by averaging the entropy of all tokens within this line, In our preliminary study, we employed this calculation method for DeepDL. However, there are some other ways to estimate the line entropy. For example, a common way is to use the max entropy as the representation. Yan et al. propose that sum average with max entropy is useful for describing the entire naturalness of the line, especially when the max entropy of different lines might be equal. To investigate the performance of our approach under different line entropy calculation methods, we combine the average entropy and max entropy as follows:

$$H_p(s) = \frac{1}{|s|}\sum_{n=1}^{|s|} H_p(t_i) + max(H_p(t_1), \ldots H_p(t_s)) \tag{12}$$

Table 8 presents the performance of our approach with respect to the three different line entropy calculating methods. From the table, we can see that the average entropy method can achieve a better performance in most of the cases, and average entropy method outperforms the other method by a large margin in average. This is the reason why we choose average entropy as the line entropy calculating method for our approach.

7.2 Time Efficiency

To analyze the complexity of our proposed model, DeepDL, we further measure the time complexity of DeepDL in terms of training and application process. Considering Yan's approach is highly efficient for JIT defect localization, we compare DeepDL with Yan's approach for time efficiency analysis. In particular, regarding the model training process, we record the time cost for training our DeepDL model and Yan's approach on each selected project respectively. Regarding the model application process, we sequentially input the commits in the test set to the model and record the time it takes to obtain the results. To reduce the bias of the experimental results, we repeat the testing process 5 times on each project and calculate the average time for processing a buggy commit. Both models are trained and tested on the same machine, which contains an Intel i9-9900k CPU and four RTX 2080Ti GPU.

Table 9 shows the results of our time cost experiment. From the table, we can observe the following points: (i) Yan's approach is highly efficient for training and testing. For example, it costs only 25s for training a project and 5ms for checking a buggy commit, this is reasonable because Yan's approach uses the n-gram language model to estimate the naturalness of the source code, which requires little computing resources and calculation time. (ii) The time cost of DeepDL is mostly for training process. DeepDL takes 1,660s on average for training a project, which is much slower than Yan's approach. This is because DeepDL is based on the neural language model, which is heavily dependent on the sizes of the source code database. However, we argue that since training DeepDL is a one-time cost, after the training process is completed, the trained DeepDL model can be easily loaded and reused. (iii) The average application time of DeepDL only costs 8ms, which means DeepDL takes 8ms on average to check a given commit. The gap between our model and Yan's approach is 3ms for application, which is difficult to notice the time difference between the two models in actual applications. Considering that checking a buggy commit using DeepDL is highly efficient, we argue that DeepDL is efficient enough for practical use.

7.3 Impact of Different Tokenizer

One of the key challenges in JIT defect localization is the out-of-vocabulary (OOV) problem. To alleviate the OOV problem, we adopt the BPE algorithm for tokenizing the source code. Two advantages can be obtained by employing the BPE algorithm as the tokenizer: (i) The BPE tokenizer can alleviate the OOV problem in the testing set, (ii) The BPE tokenizer can greatly reduce the source code vocabulary size. To quantitatively investigate the effectiveness of using

TABLE 8: The performance of different methods for calculating the line entropy

Project	Top-1 accuracy			Top-5 accuracy			MRR			MAP		
	Avg+Max	Max	Avg	Avg+Max	Max	Avg	Avg+Max	Max	Avg	Avg+Max	Max	Avg
Activemq	0.3836	0.3769	**0.4070**	0.6131	0.5829	**0.6348**	0.4939	0.4757	**0.5117**	0.4628	0.4452	**0.4749**
Closure-compiler	0.2723	0.2654	**0.2774**	**0.5257**	0.5240	0.4983	**0.3979**	0.3909	0.3933	**0.3762**	0.3677	0.3718
Deeplearning4j	0.2861	0.2539	**0.3184**	0.5771	0.5273	**0.5908**	0.4199	0.3848	**0.4464**	0.3565	0.3280	**0.3746**
Druid	0.1910	0.1573	**0.1966**	0.4466	0.4017	**0.4663**	0.3162	0.2782	**0.3297**	0.2819	0.2563	**0.2975**
Flink	0.2187	0.1913	**0.2445**	0.4351	0.3933	**0.4761**	0.3314	0.2965	**0.3584**	0.2710	0.2500	**0.2894**
Graylog2-server	0.3436	0.3333	**0.3742**	0.6450	0.6143	**0.6731**	0.4860	0.4644	**0.5077**	0.4415	0.4221	**0.4503**
Jenkins	**0.3835**	0.3485	0.3748	0.6650	0.6447	**0.6699**	0.5139	0.4861	**0.5149**	0.4776	0.4548	**0.4792**
Jetty	0.2626	0.2323	**0.2773**	0.5124	0.4830	**0.5427**	0.3840	0.3586	**0.4022**	0.3366	0.3186	**0.3537**
Jitsi	0.3627	0.3323	**0.3634**	0.6707	0.6396	**0.6798**	0.4983	0.4727	**0.5043**	0.4362	0.4186	**0.4425**
Jmeter	0.4949	0.4648	**0.4980**	0.7794	0.7502	**0.7968**	0.6219	0.5944	**0.6285**	0.5938	0.5679	**0.6048**
Libgdx	0.3432	0.3350	**0.3711**	0.6371	0.6141	**0.6568**	0.4748	0.4625	**0.4981**	0.4444	0.4265	**0.4582**
Robolectric	0.2464	0.2273	**0.2536**	0.5144	0.4665	**0.5383**	0.3762	0.3458	**0.3851**	0.3183	0.2946	**0.3301**
Storm	0.1650	0.1456	**0.1748**	**0.4078**	0.3883	0.3689	**0.2872**	0.2612	0.2789	**0.2397**	0.2200	0.2338
H2o	0.2977	0.2549	**0.3057**	0.5927	0.5294	**0.5954**	0.4345	0.3847	**0.4427**	0.3900	0.3474	**0.3996**
average	*0.3037*	*0.2799*	***0.3169***	*0.5730*	*0.5400*	***0.5849***	*0.4311*	*0.4040*	***0.4430***	*0.3876*	*0.3655*	***0.3972***
p-value	*<0.001*	*<0.001*		*<0.001*	*<0.001*		*<0.001*	*<0.001*		*<0.001*	*<0.001*	

TABLE 9: Time Cost Analysis

Project	Yan's Training Time /s	Yan's Application Time /s	DeepDL Training Time /s	DeepDL Application Time /s
Activemq	23	0.004	2437	0.009
Closure-compiler	26	0.005	1873	0.007
Deeplearning4j	30	0.005	2621	0.008
Druid	23	0.004	972	0.007
Flink	25	0.005	5312	0.008
Graylog2-server	23	0.005	1135	0.007
Jenkins	25	0.004	887	0.007
Jetty.project	26	0.006	1175	0.007
Jitsi	23	0.006	1378	0.007
Jmeter	24	0.007	687	0.008
Libgdx	27	0.005	1206	0.007
Robolectric	23	0.005	762	0.008
Storm	23	0.005	1456	0.007
H2o	26	0.008	1351	0.008
Average	*25*	*0.005*	*1660*	*0.008*

TABLE 10: The performance of different Tokenizer

Project	Vocabulary Size		OOV in Test Set	
	Traditional	BPE	Traditional	BPE
Activemq	38,900	5,581	148,879	109
Closure-compiler	42,979	5,493	127,066	265
Deeplearning4j	44,841	6,367	349,601	144
Druid	27,650	5,758	289,007	575
Flink	84,875	6,192	416,437	203
Graylog2-server	28,253	5,646	157,463	1271
Jenkins	32,296	5,855	260,513	234
Jetty.project	63,688	5,474	320,502	677
Jitsi	53,360	6,192	154,862	604
Jmeter	30,861	5,999	68,134	231
Libgdx	49,825	5,726	208,954	1435
Robolectric	32,864	6,082	211,605	537
Storm	30,086	5,427	226,368	966
H2o	47,268	5,533	451,340	538
Average	*43,410*	*5,809*	*242,195*	*556*

BPE tokenizer for solving the OOV problem, we counted the number of OOV words and the vocabulary size by using BPE tokenizer and the traditional tokenizer.

Table 10 shows the results of adopting different tokenizers for tokenizing source code. From the table, several points stand out: (i) By applying the BPE tokenizer instead of the traditional tokenizer in data processing step, the vocabulary size of the source code sharply decreases from 43,410 to 5,809 on average, which shows the ability of our approach for reducing the vocabulary size. (ii) The vocabulary size of the traditional tokenizer heavily depends on the specific project, while the BPE tokenizer can keep a relatively small vocabulary size. For example, the maximum and minimum vocabulary size of using traditional tokenizer are 84,875 and 25,253 respectively, while by using BPE tokenizer, the maximum and minimum vocabulary size are 6,367 and 5,427 respectively. This shows the robustness of BPE tokenizer for maintaining a stable code vocabulary. (iii) The advantage of using BPE tokenizer for solving the OOV problem is more obvious. For example, the average number OOV tokens in the test set is 556 by applying the BPE tokenizer, this number rockets up to 242,195 by using the traditional tokenizer. This further confirms the power of the BPE tokenizer for alleviating the OOV problem.

8 RELATED WORK

We divide our related work into three parts: defect localization, Just-in-Time defect localization and deep learning in defect prediction.

8.1 Defect Localization

8.1.1 Program Spectrum-based Techniques

A program spectrum describes the execution information of a program from certain perspectives, which can be used to track program behavior [43], [44]. Collofello and Cousins suggested that the program spectrum can be used for software fault localization [45]. Jones and Harrold [46] proposed the ESHS-based similarity coefficient-based Tarantula technique that uses a suspiciousness score which is provided by the information of successful and failed test cases to locate buggy elements. Abreu proposed the Ochiai a similarity coefficient-based technique [47]. It is generally considered more effective than Tarantula. W. Eric Wong proposed a technique using both single-fault and multi-fault programs named DStar, which outperforms Tarantula and Ochiai techniques in most cases [48]. However, spectrum-based techniques require test cases that are often unavailable [7]–[9].

8.1.2 Machine Learning-based Techniques

Li et al. proposed that it can be quite challenging for the traditional Learning-to-Rank algorithms to automatically identify effective existing/latent features. They introduce a deep learning-based approach named DeepFL to automatically learn the most effective existing/latent features for precise fault localization. The experimental results show DeepFL can significantly outperform the state-of-the-art TraPT/FLUCCS [5], [6]. Chaleshtari et al. proposed a new mutation-based fault localization approach called SMBFL to reduce the execution cost by reducing the number of statements to be mutated [49]. In the SMBFL method, the suspiciousness score of program statements is measured based on the entropy of their mutants. Recently, Lou et al. proposed a coverage-based fault localization technique, Grace, which fully utilizes detailed coverage information with graph-based representation learning [50]. But these machine learning-based techniques are employed after the defect is discovered. Our defect localization approach is applied at code check-in time.

8.2 Just-in-Time Defect Localization

Yan et al. propose a two-phase framework, i.e., Just-in-Time defect identification and Just-in-Time defect localization [12]. Especially the Just-in-Time defect localization phase is the first Just-in-Time defect localization approach. They leverage software naturalness with the N-gram model. Their model will rank the source code lines introduced by the new change according to their suspiciousness scores. The source code lines ranked at the top of the list are estimated as the defect location. They conduct an empirical study on 14 open source Java projects. Their model outperforms the PMD [51] (PMD is a commonly used static bug-finder tool and has been used in prior related studies, such as "Software defect prediction via convolutional neural network". PMD produces line-level warnings and assigns a priority for each warning.) in terms of Top-1 accuracy, Top-5 accuracy, MAP and MRR measures.

Our work is inspired by their work that locates buggy programs prior to the appearance of the defect symptoms. Yan's method use N-gram as the language model. Although they have fine-tuned the model, there are still many shortcomings of their model (e.g. containing many OOV problems, localness of the source code are not considered). Therefore, we can make more improvements in the Just-in-Time defect localization field. So we proposed a new model and the experimental results also prove that our approach outperforms Yan's.

8.3 Deep Learning in Source Code

8.3.1 Deep Learning in Defect Prediction

Deep learning algorithms have been adopted to improve research tasks in software engineering. Yang et al. propose a defect prediction model that leverages deep learning to generate new features from existing ones and then use these new features [10]. They used a Deep Belief Network (DBN) to generate features from 14 traditional change level features. Li et al. propose a framework called Defect Prediction via Convolutional Neural Network (DP-CNN) [11]. They extract token vectors based on the programs' Abstract Syntax Trees (ASTs) and feed the numerical vectors into DP-CNN to automatically learn semantic and structural features of programs. Finally, They combine the learned features with traditional hand-crafted features to predict defect. Wang et. alleverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs) [52].

Despite the above techniques being successfully used in defect prediction, there is no attempt yet at applying deep learning methods to Just-in-Time defect localization. Thus, in this paper, we leverage the deep learning based Seq2Seq model to Just-in-Time defect localization.

8.3.2 Deep Learning in Automatic Program Repair

SEQUENCER proposed by Chen et al. [53], CoCoNuT proposed by Lutellier et al. [54] and CURE proposed by Jiang et al. [55], have been developed to automatically repair source code. They all apply sequence-to-sequence model to fix bugs. However, our usage of sequence-to-sequence model is different. DEEPDL analyzes the rationality (naturalness) of the specific code line based on this line and its context and gives a risk score of suspicious buggy code lines.

8.3.3 Deep Learning in Code Representation

There are many papers on the representation of source code [17], [56]–[59]. Code2vec [56] is an example of learning distributed representations of source code. It represents snippets of code as continuous distributed vectors. Besides, Alon et al. proposed code2seq [57], which leverages the syntactic structure of programming languages to encode source code. Hoang et al. proposed CC2Vec [17], which produces a distributed representation of code changes. Although these studies completed the code representation task, they cannot complete the JIT defect location task well. DEEPDL learns the associations between clean code and its context to calculate the risk of code that contains bugs.

9 Conclusions and Future Work

We propose a novel approach, DEEPDL, to locate buggy source code lines in a defect change at check-in time. DEEPDL takes added code lines in the defect change as input. Then DEEPDL will assign a suspiciousness score to each code line and sort these code lines in descending order of this score. The source code lines at the top of the list are considered to be a possible defect location. Our experimental results show that DEEPDL outperforms the state-of-the-art approach and achieves better results in terms of four ranking measures. In future work, we plan to improve the effectiveness of our proposed approach by adding more information (e.g. source code history) to the model. We also plan to apply our proposed approach to other programming languages (e.g. C#, Python, etc). We want to evaluate our approach with developers to see if it helps them address just-in-time detected defects.

Acknowledgements

This research is supported by National Key R&D Program of China under Grant No.2019YFB1600700, ARC Laureate

Fellowship FL190100035 and the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180.

[2] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[3] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[4] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.

[5] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[6] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283.

[7] Pavneet Singh Kochhar, Ferdian Thung, David Lo, and Julia Lawall. An empirical study on the adequacy of testing in open source projects. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 215–222. IEEE.

[8] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *2013 13th International Conference on Quality Software*, pages 103–112. IEEE.

[9] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE.

[10] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE.

[11] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE.

[12] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[13] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE.

[14] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439. IEEE.

[15] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[16] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[17] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. Cc2vec: Distributed representations of code changes. *arXiv preprint arXiv:2003.05620*, 2020.

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[20] Hongyuan Mei, Mohit Bansal, and Matthew Walter. Coherent dialogue with attention-based language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

[21] Stefan Kombrink, Tomáš Mikolov, Martin Karafiát, and Lukáš Burget. Recurrent neural network based language modeling in meeting recognition. In *Twelfth annual conference of the international speech communication association*.

[22] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*.

[23] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[24] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv preprint arXiv:1608.02715*, 2016.

[25] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE.

[26] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4945–4949. IEEE.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

[28] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, and Klaus Macherey. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[29] Xin Li, Piji Li, Wei Bi, Xiaojiang Liu, and Wai Lam. Relevance-promoting language model for short-text conversation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8253–8260.

[30] Chung-Cheng Chiu, Tara N Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J Weiss, Kanishka Rao, and Ekaterina Gonina. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE.

[31] Volodymyr Mnih, Nicolas Heess, and Alex Graves. Recurrent models of visual attention. pages 2204–2212.

[32] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pages 1243–1252. PMLR.

[33] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390. IEEE.

[34] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. pages 1715–1725.

[35] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. pages 1073–1085.

[36] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[37] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[38] Frank Wilcoxon. *Individual comparisons by ranking methods*, pages 196–202. Springer, 1992.

[39] Hervé Abdi. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics*, 3:103–107, 2007.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[41] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[42] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training.

[43] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[44] Nicholas Kidd, Thomas Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 198–213. Springer.

[45] James S Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *Managing Requirements Knowledge, International Workshop on*, pages 539–539. IEEE Computer Society.

[46] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282.

[47] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE.

[48] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[49] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180.

[50] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676.

[51] Tom Copeland. *PMD applied*, volume 10. Centennial Books Arexandria, Va, USA, 2005.

[52] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE.

[53] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[54] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114.

[55] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.

[56] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[57] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[58] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 2020.

[59] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. Automating the removal of obsolete todo comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 218–229, 2021.