Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks

YI LI, New Jersey Institute of Technology, USA
SHAOHUA WANG*, New Jersey Institute of Technology, USA
TIEN N. NGUYEN, The University of Texas at Dallas, USA
SON VAN NGUYEN, The University of Texas at Dallas, USA

Bug detection has been shown to be an effective way to help developers in detecting bugs early, thus, saving much effort and time in software development process. Recently, deep learning-based bug detection approaches have gained successes over the traditional machine learning-based approaches, the rule-based program analysis approaches, and mining-based approaches. However, they are still limited in detecting bugs that involve multiple methods and suffer high rate of false positives. In this paper, we propose a combination approach with the use of contexts and attention neural network to overcome those limitations. We propose to use as the *global context* the Program Dependence Graph (PDG) and Data Flow Graph (DFG) to connect the method under investigation with the other relevant methods that might contribute to the buggy code. The global context is complemented by the *local context* extracted from the path on the AST built from the method's body. The use of PDG and DFG enables our model to reduce the false positive rate, while to complement for the potential reduction in recall, we make use of the attention neural network mechanism to put more weights on the buggy paths in the source code. That is, the paths that are similar to the buggy paths will be ranked higher, thus, improving the recall of our model. We have conducted several experiments to evaluate our approach on a very large dataset with +4.973M methods in 92 different project versions. The results show that our tool can have a relative improvement up to 160% on F-score when comparing with the state-of-the-art bug detection approaches. Our tool can detect 48 true bugs in the list of top 100 reported bugs, which is 24 more true bugs when comparing with the baseline approaches. We also reported that our representation is better suitable for bug detection and relatively improves over the other representations up to 206% in accuracy.

CCS Concepts: • **Social and professional topics** → **Professional topics**; • **Software and its engineering** → **General programming languages**; Language features.

Additional Key Words and Phrases: Bug Detection, Deep Learning, Code Representation Learning, Network Embedding, Program Graphs, Attention Neural Networks

*Corresponding Author

Authors' addresses: Yi Li, Department of Informatics, New Jersey Institute of Technology, University Heights, Newark, New Jersey, 07102, USA, yl622@njit.edu; Shaohua Wang, Department of Informatics, New Jersey Institute of Technology, University Heights, Newark, New Jersey, 07102, USA, davidsw@njit.edu; Tien N. Nguyen, Computer Science Department, The University of Texas at Dallas, 800 W. Campbell Road, Richardson, Texas, 75080, USA, tien.n.nguyen@utdallas.edu; Son Van Nguyen, Computer Science Department, The University of Texas at Dallas, 800 W. Campbell Road, Richardson, Texas, 75080, USA, sonnguyen@utdallas.edu.

2475-1421/2019/10-ART162
https://doi.org/10.1145/3360588

ACM Reference Format:
Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 162 (October 2019), 30 pages. https://doi.org/10.1145/3360588

1 INTRODUCTION

Improving software quality and reliability is a never-ending demand [Amodio et al. 2017; Bhatia and Singh 2016; Bielik et al. 2016; Hindle et al. 2012; Kim et al. 2018; Patra and Pradel 2016]. Several approaches have been introduced to help developers in detecting and fixing software defects to improve software quality [Bian et al. 2018; Cole et al. 2006; Engler et al. 2001; Jin et al. 2012; Olivo et al. 2015; Toman and Grossman 2017], ranging from static approaches (e.g., program analysis, bug detection, bug prediction, model checking, validation and verification, software mining, etc.) to dynamic approaches (e.g., testing, debugging, fault localization, etc.). Among them, bug detection helps developers to detect bugs early by scanning the source code statically and determine if a given source code is buggy [Li and Zhou 2005; Liang et al. 2016; Nguyen et al. 2009b; Pradel and Sen 2018; Wang et al. 2016a,b; Wasylkowski et al. 2007]. Bug detection has been shown to be effective in improving software quality and reliability [Bian et al. 2018; Cole et al. 2006; Engler et al. 2001; Jin et al. 2012; Olivo et al. 2015; Toman and Grossman 2017]. The existing state-of-the-art bug detection approaches can be classified into the following:

- *Rule-based bug detection.* In this type of approaches, several programming rules are pre-defined to statically detect common programming flaws or defects. A popular example of this type of approaches is FindBugs [Hovemeyer and Pugh 2007]. While this type of approaches is very effective, new rules are needed to define to detect new types of bugs.
- *Mining-based bug detection [Bian et al. 2018; Cole et al. 2006; Engler et al. 2001; Jin et al. 2012; Li and Zhou 2005; Olivo et al. 2015; Toman and Grossman 2017].* To overcome the pre-defined rules, the mining-based approaches rely on mining from existing source code. Typically, this type of approaches automatically extracts implicitly programming rules from program source code using data mining approaches (e.g., mining frequent itemsets or sub-graphs) and detects violations of the extracted rules as potential bugs. These mining-based approaches still have a key limitation in a very high false positive rate due to the fact that it cannot distinguish the cases of incorrect code versus infrequent/rare code.
- *Machine learning-based bug detection [Pradel and Sen 2018; Wang et al. 2016a,b; Wasylkowski et al. 2007].* With the advances of machine learning (ML) and especially deep learning models, several approaches have been proposed to learn from previously known and reported bugs and fixes to detect bugs in the new code. While the ML-based bug detection models [Nam and Kim 2015] rely on feature selections, the deep learning-based ones [Pradel and Sen 2018; Wang et al. 2016b] take advantages of the capability to learn the important features from the training data for bug detection. Showing the advantages over the traditional ML-based bug detection models, the deep learning-based approaches are still limited to detect bugs in individual methods without investigating the dependencies among different yet relevant methods. In practice, there exist several cases that bugs occur across more than one method. That is, to decide whether a given method is buggy or not, a model needs to consider other methods that have data and/or control dependencies with the method under investigation. Due to that, the existing deep learning-based approaches have high false positive rates, making them less practical in the daily use of software developers. For example, DeepBugs [Pradel and Sen 2018] is reported to have a high false positive rate of 32%. That is, approximately one out of 3 reported bugs is not a true bug, thus, wasting much developers' efforts. Our study (Section 4.3.1) also showed a false positive rate of 41% for DeepBugs on our dataset.

To overcome the aforementioned limitations of the state-of-the-art approaches while still taking advantage of deep learning capability, we propose a combination approach with the use of contexts and attention neural networks. In order to detect whether given methods are involved in bugs that might involve individual or multiple methods, we propose to use the Program Dependence Graph (PDG) [Ferrante et al. 1987] and Data Flow Graph (DFG) [Yourdon 1975] as the *global context* to connect the method under investigation with other relevant methods that might contribute to identifying the buggy code. The global context is complemented by the local context extracted from the path on the AST built from the method's body. The use of PDG and DFG enables our model to reduce the false positive rate when matching the given code against the buggy code in the past because two source code fragments are similar not only if their ASTs are similar but also if the global contexts in the PDG and DFG are similar. With this strategy, our model would increase its precision in detecting the buggy methods. However, to complement for the potential reduction in recall (i.e., our model might miss buggy code due to its stricter conditions on code similarity when additionally using the PDG/DFG), we make use of the attention neural network mechanism to put more weights onto the buggy paths in source code. That is, the paths that are similar to the buggy paths will be ranked higher, thus, improving recall.

Our approach works in three phases. In the first phase of building the representation for local context for buggy and non-buggy code, our model constructs the AST for a given method's body and extracts the paths along the AST's nodes to capture the syntactic structure of the source code. Prior works [Alon et al. 2018; Nguyen et al. 2009a] have shown that syntactic structure of source code can be approximately captured via the paths along their nodes with certain lengths. Word2Vec [Mikolov et al. 2013a] is used on the AST nodes along the collected paths to convert them into vectors to capture the surrounding nodes in the paths. After using Word2Vec, the generated AST node vectors are fed into an attention-based Gated Recurrent Unit (GRU) layer [Cho et al. 2014] that allows our model to encode and emphasize on the order of the nodes in a path, i.e., on the nested structures in the AST. Also, we convert the node vectors into matrices and feed them to an attention Convolutional layer [Yin et al. 2015] that allows our model to emphasize the local coherence patterns in matrices and put more weights on the buggy paths. After that, we use Multi-Head Attention [Vaswani et al. 2017] to combine the results from the attention GRU layer and attention Convolutional layer together as the path *local context* representation modeling the content of a path.

In the second phase of integrating the *global context* modeling relations among paths from methods , we build the PDG and DFG, and extract the subgraphs relevant to a method. Unlike the learning of *local context* representations for paths within an AST built from a method's body using GRU and Convolutional layers, our model uses Node2Vec [Grover and Leskovec 2016] to encode the PDG and DFG into embedded vectors to capture relations between relevant paths. Node2Vec is a widely used network embedding algorithm to convert large graphs (e.g., a PDG of a project) into low-dimensional vectors without too much graph structural information loss for efficient processing. After having both local context and global context representations for each path, we can get the representation for each method by directly linking all merged path vectors. In the last phase, we use a convolutional layer to classify the vectors into two classes of buggy and non-buggy code. Based on the results vectors, we use SoftMax to process and set up a threshold to pick the number of potential buggy methods to report for the source code under investigation.

We have conducted several experiments to evaluate our approach on a very large dataset with +4.973M methods in 92 different versions of 8 large, open-source projects. We compare our approach with the state-of-the-art approaches in two aspects. First, we compare our tool against the existing bug detection tools using *rule-based techniques* including FindBugs [Ayewah et al. 2007], *mining techniques* including Bugram [Wang et al. 2016a] and NAR-miner [Bian et al. 2018], and *deep*

learning techniques including DeepBugs [Pradel and Sen 2018]. The results show that our tool can have a relative improvement up to 160% on F-score when comparing with other baselines in the unseen project setting and a relative improvement up to 92% on F-score in the unseen version setting. Our tool can detect 48 true bugs in the list of top 100 reported bugs, which is 24 more true bugs when comparing with the baselines. Second, we compare our representation with local and global contexts against the state-of-the-art code representations that are used for deep learning models in code similarity including DeepSim [Zhao and Huang 2018], code2vec [Alon et al. 2018], Code Vectors [Henkel et al. 2018], Deep Learning Similarity [Tufano et al. 2018], and Tree LSTM [Tai et al. 2015]. We used those representations with our attention-based bug detection model and compared with the results from our tool with our representation. We reported that our representation can improve over the other representations up to 206% in F-score in the unseen project setting and up to 104% on F-score in the unseen version setting. Our tool can detect 48 true bugs in top 100 reported ones, which is 27 more true bugs when comparing with the baselines. Furthermore, we conducted experiments to study the impact of the components in our model on its accuracy. We found that while global context in the PDG and DFG improves much in Precision, Multi-head Attention mechanism helps improve much in Recall to make up for the reduction in Recall caused by the stricter condition in the PDG and DFG as the global context.

In this paper, we make the following contributions:

- **A new code representation specialized for bug detection.** To the best of our knowledge, our work is the first to learn code representation specializing toward bug detection in three novel manners: 1) directly adding weights for differentiating buggy and non-buggy paths into code representation learning; 2) combining the local context within an AST and global context (relations among paths in the PDG and DFG) for bug detection; and 3) integrating inter-procedural information using the PDG and DFG.
- **A novel bug detection approach.** We build a new attention-based mechanism bug detection approach that learns to aggregate different AST path-based code representations into a single vector of a code snippet and to classify the code snippet into buggy or non-buggy.
- **An extensive comparative evaluation and analysis.** Through a series of empirical studies, our results show that our approach outperforms the state-of-the-art ones. We also compare our learned code representation with other existing ones and the comparative empirical results show that our code representation is more suitable for detecting bugs than others. Our replication package can be found on our website [Pro 2019].

2 MOTIVATING EXAMPLE AND APPROACH OVERVIEW

2.1 Motivating Example

In this section, we will present a real-world example and our observations to motivate our approach.

Figure 1 shows an example of a real-world defect in the project named *hive* in GitHub. The bug involves three methods in which the method *getSkewedColumnNames* (method 1) retrieves the column type information from the input alias via the method *getTableForAlias* (method 3) and then compares it with the provided column names via the method *getStructFieldTypeInfo* (method 2) in order to find the suitable constant description for the current processing node. The bug occurred due to the inconsistency in handling the case-sensitivity of the names: the type field name is in the lowercase (line 2, method 2) while the alias name is not. To fix this bug, the developer added a method call to convert the alias name into lowercase (lines 2–3, method 3).

From this example, we have drawn the following observations:

Observation 1 (O1). This bug involves multiple methods. For developers to completely understand this bug or for a model to automatically detect this, it is necessary to consider multiple methods

Method 1.

```
1     public List<String> getSkewedColumnNames(String alias) {
2         ...
3         else {
4             //...
5             TypeInfo typeInfo = TypeInfoUtils.getTypeInfoFromObjectInspector(this.metaData.
6                 getTableForAlias(tabAlias).getDeserializer().getObjectInspector());
7             desc = new exprNodeConstantDesc(typeInfo.getStructFieldTypeInfo(colName), null);
8         }
9         ...
10    }
```

Method 2.

```
1     public TypeInfo getStructFieldTypeInfo(String field) {
2         String fieldLowerCase = field.toLowerCase();
3         for(int i=0; i<allStructFieldNames.size(); i++) {
4             if (field.equals(allStructFieldNames.get(i))) {
5                 return allStructFieldTypeInfos.get(i);
6             }
7         }
8         throw new RuntimeException("cannot␣find␣field␣" + field + "(lowercase␣form:␣" +
9         fieldLowerCase + ")␣in␣" + allStructFieldNames);
10    }
```

Method 3.

```
1     public Table getTableForAlias(String alias) {
2   -     return this.aliasToTable.get(alias);
3   +     return this.aliasToTable.get(alias.toLowerCase());
4     }
```

Fig. 1. A Motivating Example from the Project *hive* with Bug Report id $HIVE-60$

and the dependencies among them. This type of cross-method bugs could easily occur in practice. However, the existing ML-based bug detection approaches [Bian et al. 2018; Pradel and Sen 2018] examine the code within a method individually, without considering the inter-procedural dependencies. As an example, NAR-miner [Bian et al. 2018] derives non-association rules (e.g., if there is A, then there is no B), and uses them to detect bugs occurring in each individual method. That approach cannot detect this bug because it considers each method individually and this bug does not involve a non-association rule on the appearances of any elements. As another example, DeepBugs [Pradel and Sen 2018] uses deep learning on the names of the program entities in each method to detect bugs. DeepBugs cannot detect this bug either because it does not perform intraprocedural analysis. Moreover, the fixed method *getTableForAlias* does not contain the names similar to those of buggy methods. In fact, *toLowerCase* is a widely used API and *get* is a popular method name. A model cannot determine the error-proneness for this method by solely relying on those popular names.

Observation 2 (O2). When considering multiple methods to detect a bug, there exist multiple paths on the representations that would be used to model the methods and their interdependencies, e.g., the control flow graph (CFG), program dependence graph (PDG), or the abstract syntax tree (AST). Due to the large sheer amount of paths needed to be considered, a model should not put the same weights on all the paths. To learn from database of buggy code in the past, a model should put more weights on the buggy paths than others. However, existing approaches [Alon et al. 2018; Nguyen et al. 2009a] to represent code as graph-based embedding vectors are either putting weights on frequently occurring paths or not weighting at all. For example, code2vec [Alon et al. 2018] extracts the paths in the AST among program entities with dependencies and gives more weights to frequent paths. Those paths are used as input to a deep learning model to learn the graph-based embedding vectors for source code. Frequent paths might not be the most error-prone ones in a

program. In contrast, the buggy paths might not be the frequent ones. For example, the lines 5–7 in the method 1 are in a buggy path, however, they are not one of the frequent paths in our collected data for our experiment (which will be explained later). This is reasonable because code2vec [Alon et al. 2018] was designed for measuring code similarity, rather than for bug detection. On the other hand, Exas [Nguyen et al. 2009a] represents source code with a vector by encoding the paths with up to certain lengths in the PDG. While it is successful in code similarity at the semantic level, it considers all the paths with the same weights, thus, cannot be applied well to bug detection.

2.2 Key Ideas

With the above observations, we have built our approach with the following key ideas. First, to capture the source code containing defects, in addition to represent the body of the given code, we also use as the global context the Program Dependency Graph (PDG) [Ferrante et al. 1987] and Data Flow Graph (DFG) [Yourdon 1975]. Such graphs enable us to consider the dependencies among program entities across multiple methods, thus, enabling the representation of the buggy source code involving multiple methods. In our example, it allows our model to capture the relationships among the methods 1, 2, and 3 involving in the current bug. Specifically, it allows the connections of the important nodes such as the connection between the variable *typeInfo* at line 5 of the method *getSkewedColumnNames* (Method 1) and the parameter *alias* at line 2 of the method *getTableForAlias* (Method 3), and the connection between the variable *typeInfo* at line 7 of the method *getSkewedColumnNames* (Method 1) and the variable *field* at line 2 of the method *getStructFieldTypeInfo* (Method 2). From the two connections, the relation between the line 2 of the Method 2 and the line 2 of the Method 3 can be captured. Thus, if our model has seen a similar relation that causes a bug in the training data, it could catch this in the example.

Secondly, from Observation 2, we design an attention-based deep learning model to emphasize to learn the buggy paths and use the PDGs and DFGs as the context to capture the relations among the methods involving in a bug. The attention mechanism also helps in improving the ranking of buggy candidates, thus improving the recall that was potentially affected by the use of PDG and DFG in code matching. For example, in the history, there exists a bug that are revealed by the above connection between the line 2 of the Method 2 and the line 2 of the Method 3. By adding a weight for buggy paths with the attention mechanism, we could make all buggy paths have a higher weight than the normal paths. In our approach, we would like to only use long paths because the short paths are covered by the longer ones. In our example, the long paths can sufficiently cover the needed information between the three methods to detect the bug.

2.3 Overview of Our Approach

Let us explain the overview of our approach. To determine whether source code in a given method is buggy or not, our model relies on the following three main steps as illustrated in Figure 2:

- **Attention-Based Local Context Representation Learning**. First, our model parses the given method to build an AST. It extracts the long paths and then uses the attention GRU layer and the attention convolutional layer to build the representation for the method's body. Let us call it the local context because the paths are extracted within the given method.
- **Network-Based Global Context Representation Learning**. Second, in addition to considering the given method's body, we also encode the context of the method by building the Program Dependence Graph (PDG) and the Data Flow Graph (DFG) relevant to the method. We call them the global context because they provide the relations between the given method and other relevant methods in the project. We use the Node2Vec [Grover and Leskovec 2016] for the encoding of the PDG and DFG.

- **Bug Detection**. Finally, with the local and global contexts of the given method, we use a softmax-based classifier to decide whether the method is buggy or not.

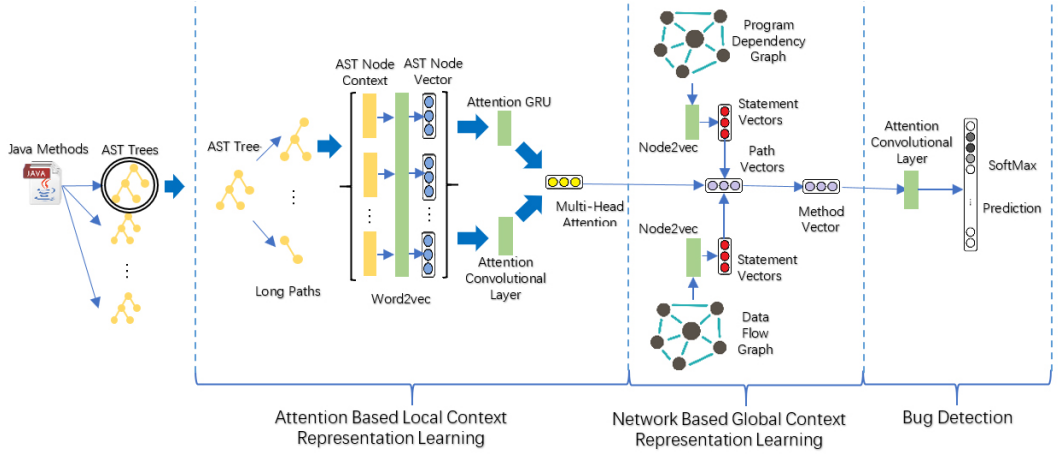


Fig. 2. Overview of our Approach.

In the step of building the local context, we choose the long paths over the AST built from the method's body. A long path is a path that starts from a leaf node and ends at another leaf node and passes the root node of the AST. As shown in previous works [Alon et al. 2018; Nguyen et al. 2009a], the AST structure can be captured and represented via the paths with certain lengths across the AST nodes. The reason for a path to start and end at leaf nodes is that the leaf nodes in an AST are terminal nodes with concrete lexemes. The nodes in a path are encoded into a continuous vector via Word2Vec and the vectors are fed into two layers: attention-based GRU layer [Cho et al. 2014] and attention Convolutional layer [Yin et al. 2015]. The GRU layer allows our model to encode and emphasize on the order of the nodes in a path, i.e., on the parent-child relationships of the AST nodes. In other words, the nested structures in an AST are captured and represented with GRU layer. Moreover, the attention-based Convolutional layer allows our model to emphasize and put more weights on the buggy paths. After that, we use Multi-Head Attention [Vaswani et al. 2017] to combine the result from attention GRU layer and attention Convolutional layer together as the path local context representation.

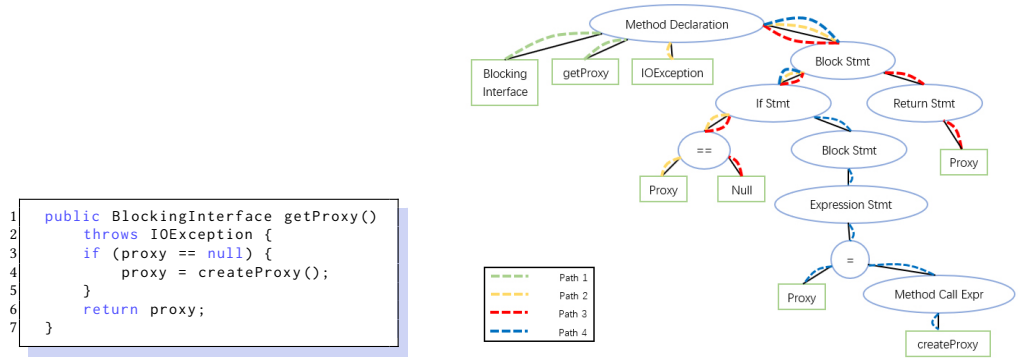
In the step of building the global context, we use the Node2Vec [Grover and Leskovec 2016] to encode the PDG and DFG into embedded vectors. These two vectors are combined by the space vectors of all nodes in each path. We use matrix multiplication and convert results together to get the path representation vector. Then, we can have method representation by appending all paths' vectors for each method. For the bug detection step, with the method vectors we can do the bug detection with a softmax-based classifier. We will explain all the steps in details next.

3 OUR APPROACH

In this section, we delve into the details of the main steps of our approach.

3.1 Attention Based Local Context Representation Learning

Given a method, we use the following steps to learn a code representation using AST paths within the method. We call it the local context as the paths used for code representation learning are within the method. Figure 4 shows the overall steps of learning local context code representation.



(a) A method example. (b) The AST of the code in Figure 3a.

Fig. 3. An example and the AST for the code from the project *hive*.

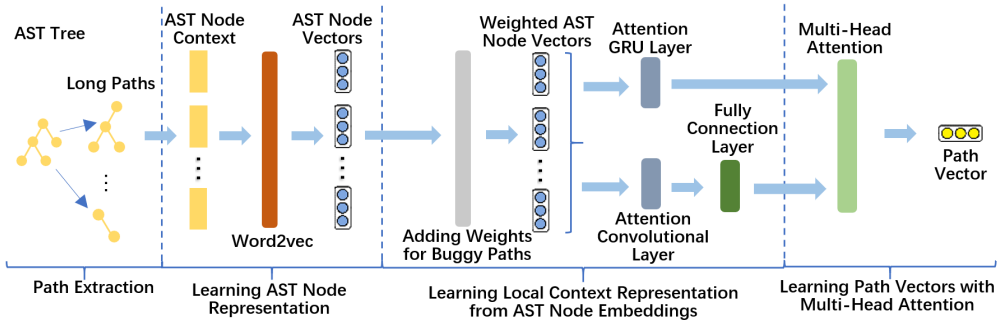


Fig. 4. The process of learning Local Context Representation

3.1.1 Path Extraction. We extract paths from an AST built for a method, instead of source code directly, because an AST helps capture better code structures. With the explicit representation of structures via ASTs, our model could make distinction better between buggy and non-buggy code structures. We use the well-known and widely-used Eclipse JDT package to build an AST for a given Java method. As we do not want to lose important information of each method for bug detection, we extract the minimum number of long paths that can cover all nodes in an AST of a method in a greedy way. In Figure 3, for the method in Figure 3a, we build an AST and extract four long paths as shown in different colors in Figure 3b. As a long path passes from a leaf node to another via the root node in an AST, there are overlapping nodes among different long paths.

3.1.2 Learning AST Node Representation. We use Word2vec [Mikolov et al. 2013a], a neural network that takes a text corpus as an input and generates a set of feature vectors for words in the corpus, to learn a vector representation for each AST node. This step takes all of the AST nodes of a method as the input, and generates a learned vector representation for each given AST node.

Specifically, in the process of AST node encoding, we treat the node content of an AST node as one word. For example, in Figure 3b, the node "Block Stm" having a real value "{}" in AST is considered as one word, *word* = "{}". Thus, we generate a sequence of words for each path. For instance, we can generate the following ordered set of words: {"Null", "==", "if()", "{}", "root","{}", "Return", "proxy"}, for the red path in Figure 3b, where "if()" is the *If Stmt*, the first "{}" right after *If*

Stmt is the *Block Stmt*, the "root" is the *Method Declaration*, the second "{}" is also the *Block Stmt* as the path passes through the root.

Given a version of a project, we extract long paths from an AST for each method and generate ordered sets of words for each path. We order the nodes in an AST path according to their appearance order in source code. We generate embeddings for each node and preserve their order. Moreover, we do not embed comments and do not differentiate specific strings and numbers during embedding, as their values are normally too specific and do not contribute to model training for bug detection.

Collectively, we obtain a large corpus of words for AST nodes from all source code under study. Each node is mapped to a word. We run Word2Vec on a large corpora to learn a vector $NodeV_i$ to represent an AST node n_i in a path of nodes $P = n_1, n_2, ..., n_i$. All nodes in training are considered to maximize the log value of the probability of neighboring nodes in the input dataset. We use Word2Vec to train our own node representations by using all of the AST nodes from each project. The loss function is defined as follows:

$$Loss_i = \min_i \frac{1}{i} \sum_{j=1}^{i} \sum_{k \in NNS^r} -\log HS\{NodeV_k | NodeV_j\}$$

$$L = \min \tfrac{1}{i} \textstyle\sum_{j=1}^{i} \sum_{k \in NNS} -\log HS\{NodeV_k | NodeV_j\} \quad (1)$$

where L is the lose function for the nodes in $P = n_1, n_2, ..., n_i$, NNS is the set of the neighboring nodes of a node n_i, and $HS\{NodeV_k|NodeV_j\}$ is the hierarchical softmax of node vectors $NodeV_k$ for the node n_k and $NodeV_j$ for the node n_j.

3.1.3 Learning Local Context Representation from AST Node Representations. After the previous step of learning AST node representation, each path, P, can be represented as an ordered set of AST node vectors, $P = \{NodeV_i, NodeV_i, \ldots, NodeV_n\}$, where n is the total number of nodes of the path, and $1 \leq i \leq n$.

To incorporate the previous buggy information into the representation learning, we add a weight to a path if the path passes an AST node that is in one or multiple bug fixes. We use the addition of weights to differentiate buggy and non-buggy AST paths. Specifically, if a node n_i from a path was in a bug fix, we apply the same weight w on all of the node vectors in the path, $P = w * \{NodeV_i, NodeV_i, \ldots, NodeV_n\}$, and the paths without any nodes in previous bug fixes have no weight added. For example, if there is a bug fix (e.g., removing the whole line) in the line 4, *proxy = createProxy()*, of the code example in Figure 3a, all of the node vectors of the blue path in Figure 3b are assigned with the same weight w.

To learn a unified vector representation from the node vectors for a path, we use two different approaches to learn path vectors capturing different aspects of key information.

[1] Attention-based GRU approach. We apply an attention-based Gated Recurrent Unit (GRU) layer [Cho et al. 2014], using an attention layer on top of the GRU layer as the weights to apply the importance on each time step during the training, to learn a path vector from a given set of node vectors. We use GRU to capture the sequential patterns from ASTs. The GRU layer, using a gating mechanism reported in [Cho et al. 2014], is a powerful and efficient model for learning a representation from a given set of vectors. There are two key gate calculations at a time step t, the reset gate r_t and the update gate z_t. The following calculation is for the j-th hidden unit at the time step t:

$$r_t^j = \sigma(W_r x_t + U_r h_{t-1})^j \quad (2)$$

$$z_t^j = \sigma(W_z x_t + U_z h_{t-1})^j \quad (3)$$

$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j \tilde{h}_t^j \tag{4}$$

$$\tilde{h}_t^j = tanh(Wx_t + U(r_t \odot h_{t-1}))^j \tag{5}$$

where x_t is the node vector for the AST node$_k$ in path$_i$ as an input of the t time step, j denotes the j-th element of a vector, the h_t^j, h_{t-1}^j is the j-th element of the output embedding vectors at the t and $t-1$ time steps, the r_t^j is the j-th element of the reset gate vector at the t time step, the z_t^j is the j-th element of the update gate vector at the t time step, σ is the logistic sigmoid function, W, U are the parameter matrices that can be learned during the training and the $\odot$ is the Hadamard product.

Given a sequence of AST node vectors of a path, we pass one node vector to the GRU layer at each time step and the GRU layer also generates an output vector at each time step. At the final time step of the GRU, one path vector is generated. Over the whole process, the GRU takes a set of node vectors as input vectors and produces a set of intermediate output vectors (i.e., the last output vector is the final generated vector). We store the set of input vectors and the set of intermediate output vectors from the GRU layer for the next step.

[2] Convolutional-based Approach. We apply an attention-based Convolutional layer in the Convolutional Neural Networks (CNN) [Cun et al. 1989], using an attention-mechanism on top of a Convolutional layer, to learn a path vector from a given set of node vectors. In the CNN, sequences of node vectors are modeled into matrices. We use the CNN to capture the local coherence patterns from the matrices of ASTs. Different node embeddings can be used to construct a matrix **D**, where it has a structure $n \times d$ of **D** with only one channel (d is the size of node embedding).

In a convolution operation, a filter can convolute a window of nodes (e.g., 3 or 4) to produce a new feature using the following equation: $c_i = \sigma(\mathbf{W} \cdot \mathbf{h}_\ell(x) + \mathbf{b})$, where c_i is the dot product of $h_\ell(x)$ and a filter weight **W**. $h_\ell(x)$ is a region matrix for the region x at location ℓ. σ is non-saturating nonlinearity $\sigma(x) = \max(0, x)$ [Krizhevsky et al. 2012]. Then the filter can convolute each possible window of nodes and produce a feature map: $\mathbf{c} = [c_1, c_2, ..., c_i, ...]$. The weight **W** and biases **b** are shared during the training process for the same filter, which enables to learn meaningful features regardless of the window and memorizing the location of useful information when appearing.

Given a sequence of AST node vectors of a path, we use all node vectors to build a $n * m$ matrix, where n is the number of node vectors and m is the length of a node vector, and send the matrix to the Convolutional (Conv) layer. At each time step, a sub-matrix is selected and used as an input for the Conv operation and the Conv layer generates an output matrix. Over the whole process, a set of matrices are built and used as inputs for the Conv layer that produces a set of intermediate output matrices (i.e., the intermediate output matrix at the last time step is the final matrix). In the Conv layer, a sequence of node vectors is transformed into a set of matrices. To reduce an obtained matrix into one dimension vector, we apply another layer: fully connected layer in the CNN [Cun et al. 1989]. We pass intermediate output matrices generated at different time steps into the fully connected layer to generate 1-dimensional intermediate vectors. We store the set of input matrices and the set of intermediate output 1-dimensional vectors from the Conv layer for the next step.

The attention-based mechanism enables our model to emphasize on the important paths that have been observed to be buggy. That is the key advantage of our model in comparison with the traditional or vanilla (standard backpropagation) GRU and CNN. While attention mechanism allows a NLP model to focus on certain important words in a sentence, it is expected to help our bug detection model to put and update more weights on the observed buggy paths.

3.1.4 Learning Path Vectors with Multi-Head Attention. In the previous step, given a sequence of AST nodes, we use two approaches to learn vector representations for a path and different vector representations capture different aspects of information of a path. To learn the unified vector for a

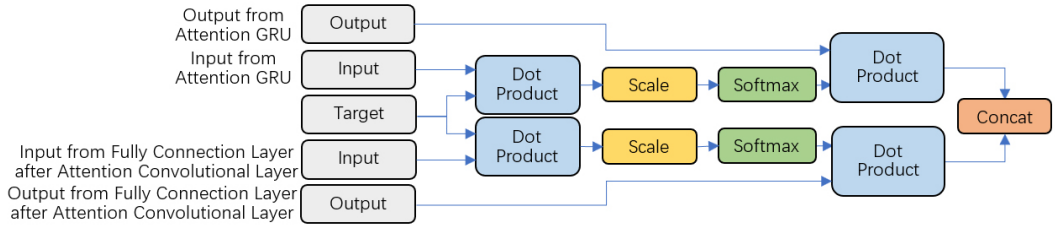


Fig. 5. Multi-Head Attention: The scale in graph is of dot products with the square root of vector dimensions.

path, we still need to find a way to combine the two path vectors into one code vector without too much information loss.

To do so, we apply the Multi-Head Attention (MHA) model [Vaswani et al. 2017] to learn a unified representation. The Multi-Head Attention model is effective in learning representation form different other representations. We build the MHA on top of the GRU and Convolutional (Conv) layers as shown in Figure 5. Due to the page limit, we only introduce the basics of MHA. In our case, we build a two-head attention and both heads have the exact same architecture, but they take different inputs. Both GRU and Conv layers take input vectors at different time steps and generate the corresponding output vectors at different time steps.

One head for the GRU layer, namely H^G, takes the following inputs: a training target vector (T), all of the input vectors of the GRU layer at different time steps, namely V_G^I, and all of the intermediate output vectors from the GRU layer at different time steps, namely V_G^O. We define a target vector to have the same length as the input vector. Also, we set all values in the T as 1 for buggy and 0 for non-buggy. V_G^I and V_G^O are both obtained in the previous step in Section 3.1.3. Then the MHA conducts a dot product between the T and the V_G^I (i.e., AST node vectors), and scale the product result by dividing it using the square root of the vector dimension of V_G^I, denoted as d. After the scale process, we apply a *softmax* function on the result of the scale process, denoted as $softmax(\frac{T \cdot V_G^I}{\sqrt{d}})$. Last, we apply the doc product operation between the result of the *softmax* operation and all of the intermediate output vectors from GRU, V_G^O. The whole process can be expressed as $V_G^p = softmax(\frac{V^T \cdot V_G^I}{\sqrt{d}}) \cdot V_G^O$, where V_G^p denotes a path vector learned from H^G.

Another head for the Convolutional layer, namely H^C, works the same way as H^G, except that H^C takes the input vectors and the intermediate output vectors from the Convolutional layer. We use V_C^p to denote a path vector learned from H^C.

The final step of MHA is to concatenate V_G^p and V_C^p to generate a unified one-dimensional path vector that incorporates local context within a method.

3.2 Network Based Global Context Representation Learning

3.2.1 Overview. As shown in Section 2.1, a bug can involve multiple methods, thus it is critical to model the relations among methods, even the paths in different methods, into code representations for bug detection. To complement the local context of buggy code, which is represented by buggy paths in the AST, we capture the global context to integrate the relations among buggy methods into our model via program and data flow dependencies. Figure 6 shows the general overview of our process to model the global context. The first step is to extract the Program Dependence Graph (PDG) and Data Flow Graph (DFG) from the source code of a Java project. In the second step, the graphs are used as the input for a process to vectorize the nodes. To achieve that, we use

the Node2Vec [Grover and Leskovec 2016] to capture the data and control dependencies between relevant program statements. Then, at the third step, the related statements in the long paths in the AST identified in the previous step for the local context are used and encoded via the representation vectors computed via the Node2Vec [Grover and Leskovec 2016]. Finally, the representation vectors for the PDGs and those for the DFGs are combined via matrix multiplication. The resulting vectors are then integrated with the vectors for the local context computed earlier to produce the path representation vectors, namely global context representation. Let us explain each step of this process in details.

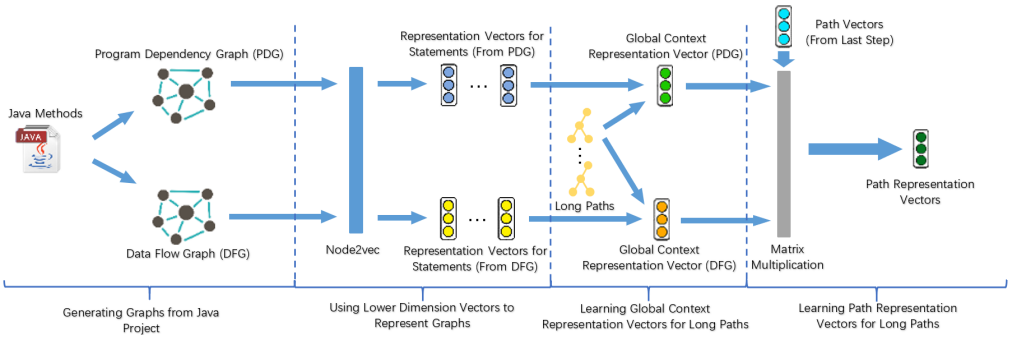


Fig. 6. Learning global Context Representation and generating Long Path Representation Vectors

3.2.2 Generating Graphs from Java Projects. As we use path-based code representation to model a method, we aim to represent the fine-grained relations among methods, i.e., the relations among paths from different methods. Specifically, we use the Program Dependence Graph (PDG) [Ferrante et al. 1987] and Data Flow Graph (DFG) [Yourdon 1975]. Given a version of a project, we generate the PDG and DFG for the entire project at the statement level. We used the Eclipse plugin Soot [Soot [n. d.]] to produce the PDG, and the plugin WALA [WALA [n. d.]] to produce the DFG.

For the PDG, we use the classes from *soot.toolkits.graph.pdg* in Soot to implement a Program Dependence Graph as defined in [Ferrante et al. 1987]. Soot can handle inter-procedural analysis for PDGs. As for the DFG, we use the classes from *com.ibm.wala.dataflow* in WALA to generate a data flow graph [Kildall 1973]. In our problem, we would like to generate a large data flow graph which contains all data flows in a whole project. Given a project, the WALA can generate a set of data flow graphs for the project and we connect them to build the entire DFG for the whole project.

Currently, for both PDG and DFG, we handle virtual method calls in a conservative way using declared types for program entities. We support no pointer analysis, data flow through heap-allocated objects is approximately and conservatively captured via explicitly declared objects.

3.2.3 Using Lower Dimension Vectors to Represent Graphs. Once the two graphs are generated for a project, we convert the large graphs (e.g., a PDG can be very large with a high number of nodes and edges) to low-dimensional vectors without much information loss for efficient processing. We apply the widely used network embedding algorithm, the Node2Vec [Grover and Leskovec 2016], to encode all of the nodes in our PDGs and DFGs. During the learning of node embeddings, the Node2Vec can encode a node with the information of the node's surrounding structures.

Technically, for each graph (i.e., the PDG or DFG), a bag of nodes is extracted in which each node m represents a code statement and the neighboring nodes of m represent the code statements with dependencies on m. A neural network in the form of a skip-gram model [Mikolov et al. 2013b] is

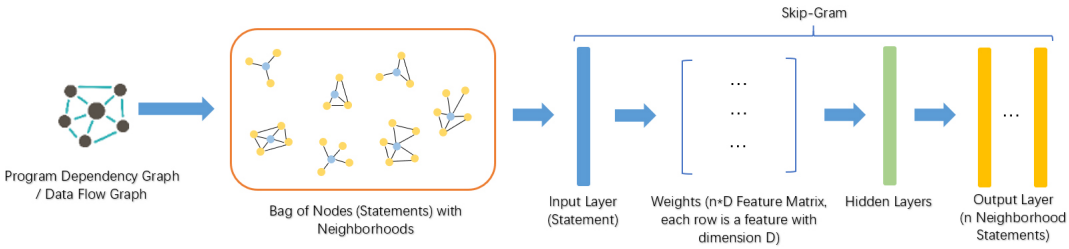


Fig. 7. Using Lower Dimension Vectors to Representation Graphs

trained with the input layer containing each node m for a statement and the output layer containing the neighboring nodes of m for the statements with dependencies on the current statement. The output of the process is the feature matrix with the dimension of $n \times D$, where n is the number of nodes/statements in the input layer and D is the number of representation features in the lower dimension vector space. In other words, each row is a feature vector representing a code statement in the input graph. The representation vectors capture the neighboring structures of the statements/nodes.

As it is not designed for source code, the Node2Vec models the network data that can flow two ways between two nodes. Our graphs, PDG and DFG, are one-directional. Therefore, we adapt the Node2Vec in the following ways. Inspired from NAR-miner [Bian et al. 2018], we set the weight $weight = 1$ to a directional edge from node A to node B if these two nodes have a dependency in a program graph (i.e., the PDG or DFG). We also assign another weight $weight = -1$ to the opposite directional edge from node B to node A, meaning that there is no dependency from B to A in a program graph. For example, in the example code in Figure 3a, there is a relationship between $if(proxy == null)$ and $proxy = createProxy()$. In the PDG, the weight on the edge from $if(proxy == null)$ to $proxy = createProxy()$ is 1, and the weight on the edge from $proxy = createProxy()$ to $if(proxy == null)$ is -1. Then, we apply the Node2Vec on the PDG and DFG, separately to compute network embedding, and obtain a vector for each node representing each code statement.

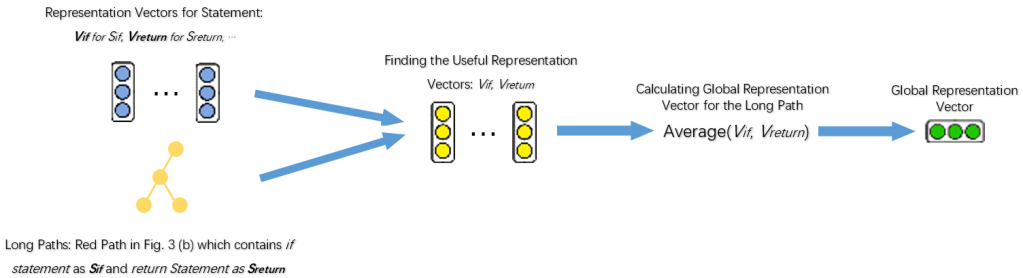


Fig. 8. Learning global Context Representation Vectors for Long Paths

3.2.4 Learning Global Context Representation Vectors for Long Paths. Once we obtain the vector representations for all of the nodes/statements in the PDG and DFG, we use them to encode the long paths in the AST that were extracted in the local context computation step. We denote a node in a PDG as N^P, and a node in a DFG as N^D. Several long paths can go through the same

statement and a statement can have multiple AST nodes. For example, in Figure 3b, the statement, $if(proxy == null)$, includes four AST nodes: *Proxy*, ==, *Null*, and *If Stmt*. There are three paths, the yellow, red, and blue paths, passing through these nodes. Therefore, given a project, we can extract all of the mappings between the long paths and the statements, where a path is mapped to a set of unique statements and each statement is a node in the graph (i.e., a PDG/DFG). The vectors representing each of those statements, i.e., the nodes in the PDG and DFG, were computed via the Node2Vec in the previous step. Thus, we can use the vectors for those nodes N^P and N^D to capture the global context and represent a long path in the AST. Specifically, a long path is represented by the average vector of the vectors for the corresponding PDG nodes (namely V^P_{PDG}) and the average vector of the vectors for the corresponding DFG nodes (namely V^P_{DFG}).

3.2.5 Learning Path Representation Vectors for Long Paths. In the final step for learning the representation vectors for long paths, we combine the local context and the global context vector representations for the paths. Given a path with a local context vector representation (in Section 3.1), denoted as V^P_{local}, we use the following steps to combine V^P_{PDG} and V^P_{DFG} with V^P_{local}: First, we apply Matrix Multiplication (denoted as $\cdot$) to V^P_{local} and V^P_{PDG}, and also V^P_{local} and V^P_{DFG}. Then we can have the path representation vector by simply merging the above two results: $V^P = V^P_{PDG} \cdot {V^P_{local}}^T, V^P_{DFG} \cdot {V^P_{local}}^T$. Matrix multiplication can effectively combine such vectors to make the combined vector more expressive. We tested other aggregation mechanisms, e.g., vector concatenation, and matrix multiplication produces better result.

Once we have all of the path representation vectors, a method M can be represented as a set of path vectors with local and global contexts: $M = \{V^P_1 \dots V^P_i \dots V^P_n\}$, where V^P_i is the unified path vector for the i-th path in M, $1 \le i \le n$, and n the total number of long paths for the method M. Within a project, each method can have a different number of long paths. To make sure all of the methods can be modeled with the same number of path vectors, we choose the method with the largest number of long paths among all methods, and use that number as the default value for the number of paths to model a method. If a method has fewer long paths than the default number, we perform zero padding for the vector representations.

3.3 Bug Detection

After the above two steps, we obtain a representation for each method. We build a classic CNN architecture to classify whether a method is buggy or not, given a set of method representations. We build the following layers to process method representations: First, a Convolutional layer is applied on the set of method representations. Second, the Max pooling and fully connected layers are applied. Third, we use a SoftMax layer to the classification.

4 EMPIRICAL EVALUATION

4.1 Research Questions

We have conducted several experiments to evaluate our model. Specifically, we seek to answer the following research questions:

RQ1. Bug Detection Comparative Study. How well our approach perform in comparison with the existing state-of-the-art bug detection approaches?

RQ2. Code Representation Comparative Study. Is our code representation with local and global contexts more suitable than existing code representations in bug detection?

RQ3. Sensitivity Analysis. How do various factors affect the overall performance of our approach?

4.2 Experimental Methodology

Table 1. Statistics on Dataset

Project Name	# of Versions	# of Files	# of Methods	# of Buggy Methods
pig	9	8k	95k	21k
avro	7	2k	30k	1k
lucene-solr	14	93k	1.032M	518k
hbase	9	14k	318k	258k
flink	14	49k	419k	173k
hive	18	53k	981K	411k
cloudstack	11	55k	766k	307k
camel	10	172k	1.332M	135k
Total	92	402K	4.973M	1.824M

4.2.1 ***Data Collecting and Processing.*** We conduct our study on eight well-known and large open-source Java projects with different versions of each project. In total, we got 92 versions of these projects with +4.9 million Java methods (Table 1). For each project, we collect source code and commits from the Github repository, and bug reports from the issue tracking system of the project. Specifically, we use the following steps to process the data of a project:

- First, we download all bug reports that are marked as *resolved or closed* and *bug* from the JIRA issue tracking system. The details of a report has a field named *Fix Versions* which indicates the bug fix locations. We extract the bug id and the version numbers from a bug report.
- Second, for a version of a project, we download the commits from the Git repository. We use the same approach as the ones used in [Mockus and Votta 2000; Ray et al. 2016, 2014] to process each commit message and mark it as a *bug-fix* if the message contains a bug id and at least one of the error related keywords: {error, bug, fix, issue, mistake, incorrect, fault, defect, flaw and type}.
- Once we identify all of the bug-fixes in the previous steps, we download the source code of the project version as a clean version and use the bug-fix commits to recover the *buggy* version from the source code, as a code commit records all of the additions and deletions. Specifically, we use the additions in commits to locate the methods where code fixes occurred, and then roll the methods back to the states before the fixes to obtain the buggy code.

4.2.2 ***Experiment Setup and Procedure.*** To answer our research questions, we use the following procedures and setups.

RQ1. (Bug Detection Comparative Study) Analysis Approach.

Comparable Baselines. We compare our approach with the following state-of-the-art approaches:

- **DeepBugs**[Pradel and Sen 2018]: DeepBug is a bug detection approach with a deep learning model on the name-based information in source code. We used the default values of their model and tried different values for vocabulary size, and kept the value giving the best result.
- **Bugram**[Wang et al. 2016a]: Bugram uses *n*-gram to evaluate a given method and decides its bugginess by picking the top-ranked possibilities for each n-gram. With our dataset, we tried various values of *n* and sequence lengths, and kept the ones giving best results.
- **NAR-miner**[Bian et al. 2018]: NAR-miner mines negative rules on the code (e.g., if A then not B), and then use them to detect bugs. We used the default values for their model.

- **FindBugs**[Ayewah et al. 2007]: FindBugs is an open-source static analysis tool that analyzes Java class files to detect program defects. The analysis engine statically encodes more than 300 different bug patterns using a variety of techniques. We used the default values for FindBugs.

We conduct our experiments in two settings:

- **Detecting bugs in unseen projects (Cross-Project).** This setting is used to test the ability of a model to detect bugs on unseen projects (i.e., on a project that is not included in the training data). We train a model on all of the versions of 7 randomly chosen projects in our dataset, and test the trained model on the remaining project. We repeat 8 times for cross validation and calculate the average.
- **Detecting bugs in unseen versions of a project.** This setting is used to test the ability of a model trained on all of the existing versions of a project and other projects to detect bugs on an unseen version of a project. We use all of the versions of 7 randomly picked projects and all of the previous versions of the 8th project as the training data and use the newest version of the 8th project as the testing data.

Qualitative Analysis. In this experiment, for comparison, we make qualitative analysis by comparing the top 100 results that each model reported on the same randomly chosen version of a project. We manually verified each reported bug. We computed how many true bugs each model can detect in top 100 results, how many true bugs detected by the baseline models were covered by our model, how many bugs our model did not cover, and how many new bugs our model can find when comparing with the baseline models.

Tuning our approach and the baseline models. We tuned approaches in the cross-project setting. For simplicity, we use the same set of parameter settings for approaches in both of the above mentioned experimental settings once the best settings are identified.

We tuned our model with the following key hyper-parameters:

1. Epoch size (i.e., 100, 200, 300),
2. Batch size (i.e., 32, 64, 128, and 256),
3. Learning rate (i.e., 0.005, 0.010, 0.015),
4. Vector length of word representation and its output (i.e., 150, 200, 250, 300),
5. Convolutional core size (i.e., $1x1$, $3x3$, $5x5$), and
6. The number of convolutional core (i.e., 1, 3, and 5).

We tuned the baseline models with some parameters to obtain the best results on our dataset. We tuned the vocabulary size for the *DeepBugs*, the gram size, sequence length range, minimum token occurrence, and reporting size for the *Bugram*, and the threshold of frequent itemsets, the maximum support threshold of infrequent itemsets, the minimum confidence threshold of interesting negative rules min_conf for *NAR-miner*. FindBugs is a rule-based tool and we directly used its default setting.

RQ2. (Code Representation Comparative Study) Analysis Approach.

In this work, for bug detection, we introduce a novel path-based code representation with graph-based local and global contexts. We aim to compare our representation with other baseline representations that are used for source code similarity in the context of bug detection.

Comparable Baselines: We compare our code representation with the following state-of-the-art code representations on bug detection:

- **DeepSim**[Zhao and Huang 2018]: DeepSim represents source code for code similarity. It encodes control flows and data flows into a semantic matrix in which each element is a high dimensional sparse binary feature vector.
- **code2vec**[Alon et al. 2018]: code2vec uses most frequently paths on the AST from one leaf node to another via going up in the tree.

- **Code Vectors**[Henkel et al. 2018]:The approach uses abstractions of traces obtained from symbolic execution of a program as a representation for learning word embedding.
- **Deep Learning Similarity** (DL Similarity) [Tufano et al. 2018]: This approach applies deep learning on 4 different types of representations, including Identifiers, AST, Control Flow Graph, and Bytecode of a method, to learn a code representation.
- **Tree-structured LSTM** (Tree-based LSTM) [Tai et al. 2015]: Tree-based LSTM gets the representation of each method by training a tree-structured LSTM model with the AST of the source code.

As those representations are not aimed for bug detection, to be fair, we compare only the code representation part of our approach with those representations. To do so, we build a baseline as follows: *we use each of those code representations with our model to detect bugs.*

Similar to the analysis approach for RQ1, we also conduct our experiments in the two settings cross-project and cross-version, and the top-ranked qualitative analysis.

Tuning our model and the baselines: As in RQ1, we use the same set of parameter settings of an approach for both experimental settings. For our approach, we use the best parameter settings learned in RQ1 to run our approach in this experiment. For the baselines, there are no key parameters in the code representation learning of the baselines.

RQ3. Sensitivity Analysis Approach.

For sensitivity analysis, we would like to evaluate how various factors including paths over AST, multi-head attention, Program Dependency Graph, and Data Flow Graph impact on our model's accuracy in bug detection. To perform sensitivity analysis, we add each element into the model one by one in each of the two settings with different parameters.

4.2.3 ***Experiment Metrics.*** We use the following metrics to measure the effectiveness of a model:

$$Recall = \frac{TP}{TP + FN},\ Precision = \frac{TP}{TP + FP},\ F\_score = \frac{2 * Recall * Precision}{Recall + Precision}$$

Where TP = True Positives; FP = False Positives; FN = False Negatives; TN = True Negatives.

Recall measures how many of the labeled bugs can be correctly detected, while Precision is used to measure how many of the detected bugs are indeed labeled as a bug in the bug tracking system. Note that in the bug tracking system, there exist cases in which the occurrences of the labels or bug-indicating words do not really show bug fixes. Thus, to complement for that, in the comparative study, we picked 100 results and manually verified if they are truly bugs or not.

In our qualitative analysis, we also computed *related Recall, Precision, and F-score*. We use the term "related" to refer that we only consider the top 100 results from a model. For related Recall, we collect all true bugs found by a model in the top 100 results and regard them as the total true bugs. We calculate the related Recall in the top 100 results with the total true bugs. For related Precision, we regard top 100 results as the total reported results and calculate the Precision with 100 results. Thus, $TP + FP$ is equal to 100. For related F-score, we use related Recall and related Precision for the calculation in the same way as in F-score.

4.3 Experimental Results

4.3.1 Results of RQ1 (Comparative Study on Bug Detection). As seen in Table 2, **our model outperforms the state-of-the-art bug detection baselines in the cross-project setting**. Specifically, our model improves over the baselines in every measurement metric, except the recall values in the rule-based approaches NAR-miner and FindBugs. The Recall values of NAR-miner and FindBugs are 6% and 12% higher than ours. However, NAR-miner and FindBugs have a high false positives rate, i.e., 52% and 66%, that is approximately 1.5x and 2.1x higher than ours. False positives

waste developers' time in investigating incorrect cases. **Our model improves F-score over the baselines DeepBugs, Bugram, NAR-miner, and FindBugs by 1.38x, 1.16x, 2.63x, and 3.57x, respectively**. Importantly, our model can generate fewer false positives than all of the baselines.

Table 2. RQ1. Comparison with the Baselines in the Cross-Project Setting. FP: False Positives

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
Recall	0.68	0.62	0.64	0.72	**0.76**
Precision	**0.39**	0.25	0.32	0.11	0.08
F-score	**0.50**	0.36	0.43	0.19	0.14
FP Rate	**0.21**	0.41	0.39	0.52	0.66

Table 3. RQ1. Comparison with the Baselines in Detecting Bugs in Unseen Versions of a Project.

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
Recall	**0.74**	0.63	0.67	0.68	0.73
Precision	**0.56**	0.27	0.41	0.22	0.15
F-score	**0.64**	0.38	0.51	0.33	0.25
FP Rate	**0.29**	0.37	0.38	0.35	0.43

Table 3 shows that **our model outperforms four state-of-the-art bug detection baselines in detecting bugs in the unseen versions of a project**. Specifically, our model relatively improves DeepBugs, Bugram, NAR-miner, and FindBugs by 107%, 37%, 155%, and 273% respectively, in terms of Precision, and by 69%, 25%, 92%, and 156% respectively, in terms of F-score.

Furthermore, consolidating the results in Table 2 and Table 3, we can see that including previous versions of a project in the training can improve the overall effectiveness of all approaches. This is reasonable because including previous versions of the same project increases the knowledge to train the model. Our model obtains a large gain in terms of F-score (i.e., increasing 0.13 from 0.51 to 0.64), which shows that our model has a better learning ability in both settings. Although NAR-miner also obtains a large gain in terms of F-score, its F-score is still very low, i.e., 0.33.

Qualitative Analysis of RQ1. Table 4 and Figure 9 show the overlapping analysis on the 100 top-ranked results from all of the models. As seen, our model discovers more true bugs than all other baselines. Specifically, 69%, 74%, 71% and 69% of the true bugs detected by DeepBugs, Bugram, NAR-miner, and FindBugs, respectively, can also be detected by our model. Although DeepBugs, Bugram, NAR-miner, and FindBugs can detect 8, 9, 7, and 4 true bugs that our model cannot detect, our model detects 30, 23, 31, and 39 more new true bugs than those baseline models, respectively. The key reason for our model to detect more bugs than DeepBugs and Bugram is that it combines both local and global contexts, along with program dependencies. On the other hand, the rule-based approaches, i.e., NAR-miner and FindBugs, are less flexible than our model because their rule-based detection engine is strict and cannot match the bugs that are not encoded in their dataset of rules.

4.3.2 Results of RQ2 (Code Representation Comparative Study). Table 5 shows that **our code representation with local and global contexts is more suitable than other existing code representations in bug detection in the cross project setting**. Specifically, our approach can outperform the five baselines using different code representations in every measurement metric, except that Recall of Tree-based LSTM code representation is 21% higher than ours. However,

Table 4. RQ1 Qualitative Analysis on the Top 100 Reported Bugs of the Approaches.

Category	Our Approach	DeepBugs	Bugram	NAR-miner	FindBugs
# of True Bugs	**48**	26	34	24	13
Related Recall	**0.70**	0.38	0.49	0.35	0.19
Related Precision	**0.48**	0.26	0.34	0.24	0.13
Related F-score	**0.58**	0.32	0.41	0.29	0.16

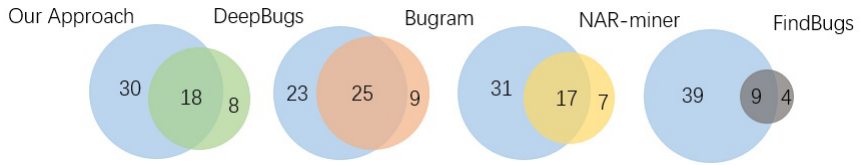


Fig. 9. Overlapping Among Results from Our Model and the Baselines in RQ1

Table 5. RQ2. Comparison with the Baselines in the Cross-Project Setting.

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-based LSTM	Code Vectors
Recall	0.68	0.67	0.71	0.69	**0.82**	0.70
Precision	**0.39**	0.19	0.24	0.17	0.09	0.15
F-score	**0.50**	0.30	0.36	0.27	0.16	0.25
FP Rate	**0.21**	0.35	0.36	0.43	0.69	0.45

Table 6. RQ2. Comparison with the Baseline Code Representations in Detecting Bugs in Unseen Versions

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-based LSTM	Code Vectors
Recall	**0.74**	0.69	0.57	0.64	0.61	0.59
Precision	**0.56**	0.42	0.33	0.41	0.21	0.25
F-score	**0.64**	0.52	0.42	0.50	0.31	0.35
FP Rate	**0.29**	0.38	0.34	0.39	0.44	0.41

Tree-based LSTM has much lower Precision (i.e., 9%) and higher false positives rate (i.e., 69%) that is 2.29X higher than our false positives rate, thus, making Tree-based LSTM impractical.

Our model using path-based code representation with local and global contexts can improve relatively over all of the five baseline code representations: DeepSim, DL-similarity, code2vec, Tree-based LSTM, and Code Vectors by 67%, 38%, 82%, 206%, and 101%, respectively, in terms of F-score. Importantly, our false positives rate is lower than all of the baselines.

Table 6 shows that **our code representation is also more suitable than other existing code representations in detecting bugs in unseen versions of a project by including other previous versions of the project in training data.** Our approach can outperform all of the baseline code representations in every measurement metric. Overall, the effectiveness of all approaches on

detecting bugs in an unseen version of a project can be improved by adding more previous versions of the project into training.

Qualitative Analysis of RQ2. We conduct the overlapping analysis on the results from all of the models. Table 7 and Figure 10 show that our model detects 64%, 79%, 67%, 67%, and 65% of the bugs that DeepSim, Deep Learning Similarity, code2vec, Tree-based LSTM, and Code Vectors detect, respectively. Although the five baselines can detect some true bugs that ours cannot detect, our approach can detect 32, 21, 22, 34, and 33 new true bugs that the code representations DeepSim, DL similarity, code2vec, Tree-based LSTM, and Code Vectors cannot detect. Thus, our representation is more suitable than the baselines in bug detection in the cross-project setting.

Table 7. RQ2. Qualitative Analysis on the Top 100 Reported Bugs of Each Model.

Category	Our Approach	DeepSim	DL Similarity	code2vec	Tree-based LSTM	Code Vectors
# of True Bugs	**48**	25	34	39	21	23
Related Recall	**0.65**	0.34	0.46	0.53	0.28	0.31
Related Precision	**0.48**	0.25	0.34	0.39	0.21	0.23
Related F-score	**0.55**	0.29	0.39	0.45	0.24	0.26

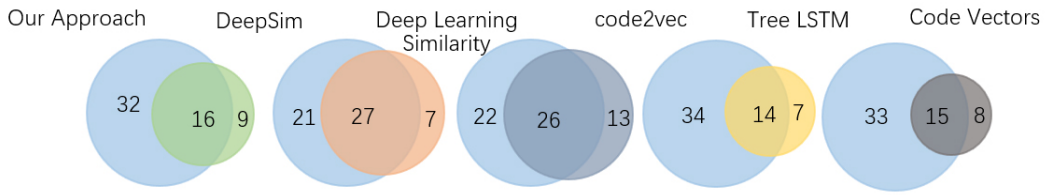


Fig. 10. Overlapping Among Results from Our Model and the Baselines in RQ2.

Table 8. RQ3. Sensitive Analysis of the Impact of Different Factors on Our Approach. LC: Local Context. PDG: Program Dependency Graph. DFG: Data Flow Graph. Attention: Multi-head Attention.

Models	Precision	Recall	F-Score
LC	0.19	0.75	0.30
LC+Attention	0.2	0.75	0.32
LC+PDG	0.29	0.54	0.38
LC+PDG+Attention	0.28	0.71	0.40
LC+DFG	0.26	0.51	0.34
LC+DFG+Attention	0.26	0.66	0.37
LC+PDG+DFG	0.37	0.43	0.40
LC+PDG+DFG+Attention	0.36	0.62	0.46

4.3.3 Results of RQ3 (Sensitivity Analysis). We conducted an experiment to study how different factors including the Local Contexts, the Multi-Head Attention, the Program Dependency Graph

and Data Flow Graph (both graphs are global contexts) affect our model's accuracy. Due to the page limit, we report the results in the cross-project setting.

As shown in Table 8, we build 8 variants of our approach with different factors and their combinations. We analyze our results in Table 8 as follows:

From *LC* in Table 8, we can see that by using only local contexts, i.e., context in individual Abstract Syntax Tree, to model source code, our model can achieve a high recall of 0.75.

To study the impact of the PDG, we compare the results obtained from two variants: *LC* and *LC+PDG*. The results show that adding the PDG as a context can increase Precision and F-score relatively by 52.6% and 26.7%, but decrease Recall from 0.75 to 0.54 (i.e. 28%). This is because the PDG puts a stricter condition on source code similarity.

To study the impact of DFG, we compare the results from two variants: *LC* and *LC+DFG*. The results show that adding the contexts in DFG can increase Precision and F-score by 36.8% and 13.3%, but decrease Recall from 0.75 to 0.51 (i.e., 32.0%). Overall, adding global contexts can improve Precision, but hurts Recall, which reduce the false positives. However, the overall F-score is improved using either of the two graphs. From the above results, we can see that the PDG can contribute more than DFG. This is reasonable because the PDG contains richer information than DFG.

To study the impact of Multi-Head Attention, we compare the results from the variants *LC* and *LC+Attention*. We can see that Multi-Head Attention cannot help much if we consider only the paths within the method's body because the attention is aimed to put weights on the global context via the PDG and DFG. However, if we compare *LC+PDG* and *LC+PDG+Attention*, we can see that Recall is improved from 0.54 to 0.71. Similar trends can be observed when we compare *LC+DFG* and *LC+DFG+Attention*, or *LC+PDG+DFG* and *LC+PDG+DFG+Attention*. This implies that **Multi-head Attention contributes much in term of improving Recall** because it helps better ranking of the buggy methods in the resulting list.

To compare *LC+PDG+DFG* with *LC+PDG* and with *LC+DFG*, we can see that both of the graphs in the global context contributes positively on our model's accuracy. When we put together local context in LC, global context in the PDG and DFG, and attention, our model achieves the highest accuracy in all metrics.

Local context enables a high recall. Global context in the PDG and DFG improves much in Precision and reduces false positive rates due to the stricter similarity condition with the use of the PDG and DFG. However, it hurts Recall. To make up for such reduction in Recall, Multi-head Attention mechanism emphasizes on the buggy paths, and helps collect more potential buggy methods, and push the buggy methods to be ranked higher in the resulting list. Thus, **Multi-head Attention mechanism helps our model make up for the reduction of Recall. As a result, F-score of our model is improved.**

5 DISCUSSION AND IMPLICATIONS

5.1 In-depth Case Studies

Let us present in-depth case studies to understand why our attention neural network-based approach using local and global contexts for learning code representations achieves better results than other approaches. Let us illustrate via the following case studies.

Case Study 1. This case study shows a typical example of a bug involving multiple interdependent methods. Figure 11 shows a bug-fix example involving two methods of the project *Camel*, for the bug with an id *Camel* − 12228. The bug report states that Method 1 *print(Doc doc, int copies, boolean sendToPrinter, String mineType, String jobName)* has a bug causing *"print command fails in case of multiple copies"*, as it requires to cancel the loop of print and reduce one parameter *int copies*. The second method *print(InputStream body, String jobName)* has a call (line 6) to Method 1.

Method 1.

```
1  -   public void print(Doc doc, int copies, boolean sendToPrinter, String mimeType,
2          String jobName) throws PrintException {
3  +   public void print(Doc doc, boolean sendToPrinter, String mimeType, String jobName)
4          throws PrintException {
5          ...
6      }
```

Method 2.

```
1
2      private void print(InputStream body, String jobName) throws PrintException {
3          if (printerOperations.getPrintService().isDocFlavorSupported(printerOperations.
4              getFlavor())) {
5              PrintDocument printDoc = new PrintDocument(body, printerOperations.getFlavor());
6  -           printerOperations.print(printDoc, config.getCopies(), config.isSendToPrinter(),
7                  config.getMimeType(), jobName);
8  +           printerOperations.print(printDoc, config.isSendToPrinter(), config.getMimeType(),
9                  jobName);
10         }
11     }
```

Fig. 11. Case study 1: The code fixes are from *Camel version* 2.20.0 for the bug *Camel* – 12228. The errors are marked in red. For simplicity purpose and page limitation, we only show the key lines of fixes.

Therefore, to fix bug id *Camel* – 12228, it requires to fix both methods. Both methods are identified as buggy by our model, but not by the baselines, which consider only individual methods.

The capability to detect this type of popular bugs involving multiple methods is due to the way we model the **relations among paths in the ASTs of a project into code representation**. In the process of code representation learning, we use the dependencies of the entities in the PDG and DFG to capture the relations among paths from the ASTs of a project. In this way, when analyzing the contexts in the AST paths and their relations of the above two methods, our model, trained with existing bug knowledge, syntax, and dependencies from the program entities, can learn to decide that both methods are buggy.

Other baselines, such as MAR-miner, Bugram, and DeepBugs, do not consider the relationships among methods and paths from a perspective of a whole version of a project. They often consider methods individually. In the above example, if a model looks only at the second method *print(InputStream body, String jobName)* without analyzing the dependencies among AST paths from both methods, it cannot detect a bug in this method.

Case Study 2. Figure 12 shows an example of two real bug fixes on the same method *main*() in two versions 0.2.0 and 0.8.0 of the project named *pig*. The method *main()* in the version 0.2.0 was fixed before. However, the fix (line 3, Method 3 in version 0.3.0) was marked as a bug in the version 0.8.0. By including the previous fixes in the version 0.3.0, our approach is able to identify the *main()* in the version 0.8.0 as buggy, while the baselines cannot.

Extracting long paths and using attention model to add weights in previous buggy paths into code representation. Our model adds weights to previous buggy paths and extracts long AST paths to cover each node in the ASTs to make sure that key information on the bug in the methods to be considered, which makes our code representation more *specialized for bug detection*. In the case study 2, we can see that when the method *main*() was fixed in the version 0.2.0, the AST paths covering the line 3 (i.e., the buggy line) will be given a weight and our model automatically learns the value for the weight based on a large number of previous fixes. Once the weight is learned, our model can learn to analyze the same or similar AST paths as buggy with higher possibilities. Thus, our model can classify the method in the version 0.8.0 as buggy based on the bug in the earlier version. In addition, our model uses long paths to cover all contexts, such as

Method 3 in version 0.2.0

```
1       public static void main(String args[])
2           ...
3   -       pigContext.getProperties().setProperty("pig.logfile", logFileName);
4   +       if(logFileName != null) {
5   +           log.info("Logging error messages to: " + logFileName);
6   +       }
7           ...
8       }
```

Method 3 in version 0.8.0

```
1       public static void main(String args[])
2           ...
3   -       if(logFileName != null) {
4   -           log.info("Logging error messages to: " + logFileName);
5   -       }
6           pigContext.getProperties().setProperty("pig.logfile", (logFileName == null?
7               "": logFileName));
8           configureLog4J(properties, pigContext);
9   +       if(logFileName != null) {
10  +           log.info("Logging error messages to: " + logFileName);
11  +       }
12          ...
13      }
```

Fig. 12. Case study 2. The code fixes are from project *pig version* 0.2.0 and *version* 0.8.0 for the bugs, *PIG* − 695 and *PIG* − 1407. The errors are marked in red and the fixes are highlighted in green. For simplicity purpose and page limitation, we only show the key lines of fixes that affect both methods.

nodes and their relations in ASTs. In the above example, the long paths help our model cover the buggy line (line 3 of the method in the version 0.2.0). However, the baselines consider only some portion of AST contexts. For example, *code2vec* considers only the most-frequent AST paths (i.e., no buggy information is considered in code2vec). In case study 2, *code2vec* assigns a weight of 0.3 to the paths covering the buggy line (line-3). However, *code2vec* does not consider the paths with a weight of 0.3 as a top ranked path, so it misses the buggy path. Also, all other baselines, such as *code2vec, Tree-based LSTM, Code Vectors, and DL Similarity*, do not incorporate buggy information in their code representation learning, which makes them miss important information in bug detection.

Attention models help add weights to buggy paths in learning code representation, thus improve the ranking, leading to improve Recall. *Attention GRU layer* and *Attention Convolutional layer* extract different types of key information from the AST in a method. Second, we use a powerful multi-head attention model [Vaswani et al. 2017] to combine the key information from different attention layers. The baselines only concatenate different vectors into a unified vector without learning, which may contain less information than our approach in code representations.

Learning to detect bugs, rather than memorization. We checked AST path duplication in training and testing data and found that 26.1% of paths in testing are in training. Our model achieves the precision of 39% (i.e., higher that 26.1%), proving that our model is able to learn from data to detect bugs, rather than simply memorization and retrieving from the stored data.

5.2 Time Complexity

Table 9 shows that all deep learning based approaches take more time to train, which is well expected. The models can be trained off-line, so the detection time is more important. On average, our approach uses 4 minutes to finish detecting bugs in a project. Although our model costs more time in detecting bugs in a project than other baselines except for the FindBugs, there is only 1 or 2 minutes difference between our model and the baselines due to the time complexity of handling graphs. However, our model performs much better than the baselines on detecting bugs. Due to

Table 9. Time Consumptions in Minutes of Approaches in Training and Detecting Bugs from a Project under the setting of detecting bugs in unseen versions of a project. DB: DeepBugs. BR: Bugram. NAR: NAR-miner. DS: DeepSim. DLS: Deep Learning Similarity. c2v: code2vec. TL: Tree LSTM. CV: Code Vectors. FB:FindBugs. N/A:not applicable.

Time	Ours	DB	BR	NAR	DS	DLS	c2v	TL	CV	FB
Training/Mining Time	654	238	6	2	497	428	125	219	246	N/A
Detection Time	4	2	2	1	2	3	2	2	2	5

the page limit, we report only the running time of the models in the setting of detecting bugs in unseen versions of a project. The time complexity evaluated in the cross-project setting is roughly similar as the ones in Table 9.

5.3 Limitations of Our Approach

Through analysis on the bugs that our model cannot detect, we identify the following limitations:

- *Our approach does not work well on the bugs about parameters in loops.* Our approach examines all of the contexts in paths and their relations, but our AST path-based modeling cannot accurately capture the relations among parameters in a loop due to the limitations of our static analysis approach. Dynamic analysis on execution paths could be useful to complement with our approach.
- *Our approach does not work well on the bugs about the fixes in strings.* Our approach does not analyze the semantics of string literals and variables, so we cannot detect the bugs that is relevant to changes in string literals. We found that NAR-miner performs better on this kind of bugs by generating the negative rules to pick out the buggy words in the strings.

5.4 Explanation Ability

To improve the ability to explain the buggy code in our solution, we could improve our solution in the following two directions in the future:

- Statement-level bug detection. Code statement-level bug detection is a natural next step of method-level bug detection. In order to detect buggy code lines, on the top of method information, we plan to utilize and incorporate the following information related to a code statement, *cs*: (1) sequential information of characters and tokens in *cs*; (2) *cs*'s relations with other code statements within one code method; and (3) *cs*'s relations with other relevant code statements from other code methods. Based on the above information, we plan to build code representations for code statements and propose deep learning based approaches to classify code statements.
- Fine-grained bug classification. In this paper, our focus is to determine whether a method is buggy or non-buggy. In the future, we plan to show the types of bugs associated with a detected code statement or method. To do so, we first plan to study bugs collected in our big dataset and manually create a small well-classified dataset of bugs. Next, we will explore to develop deep learning and active learning based approaches to automatically label more bugs using the small dataset of bugs as a seed. Last, we can develop and train machine learning (including deep learning) models on the built large dataset containing code and bug types to conduct more explainable bug detection.

6 RELATED WORK

Here, we summarize some studies relevant to our study.

6.1 Bug Detection

Many techniques have been developed for rule-based and learning-based bug detection. Some existing rule-based bug detection approaches, such as [Bian et al. 2018; Cole et al. 2006; Engler et al. 2001; Jin et al. 2012; Olivo et al. 2015; Toman and Grossman 2017], are unsupervised and very efficient. However, new rules are needed to define to detect new types of bugs, for example, in FindBugs [Hovemeyer and Pugh 2007]. The mining-based approach in NAR-miner [Bian et al. 2018] extracts negative rules to detect bugs and outperforms rule-based approaches that are based on mining positive code rules. Our experiment results show that it only costs NAR-miner 1 minute to perform bug detection on a project. However the mining-based approaches mainly suffer the problem of high false positive (FP) rates, such as the NAR-miner has a high FP rate, i.e., 52% in the cross-project setting, which make them impractical for daily use. When comparing our approach with the existing state-of-the art rule-based and mining approaches, the main differences are as follows. First, we consider the relations among paths from different methods for detecting cross-method bugs. However, they normally work on individual methods and cannot work well on the cross-method bugs. Second, our approach covers path information in an AST in order to detect very detailed bugs in each method, while the they often consider the important rules and may miss some information outside of their rules.

There exist machine learning-based bug detection approaches, including the deep learning techniques [Pradel and Sen 2018] and traditional machine learning techniques [Engler et al. 2001; Li and Zhou 2005; Liang et al. 2016; Wang et al. 2016a,b; Wasylkowski et al. 2007]. For example, the Bugram [Wang et al. 2016a] uses n-gram models to rank the methods and then picks the top-ranked methods as buggy methods. DeepBugs [Pradel and Sen 2018] uses deep learning techniques to propose a name-based bug detection approach. In this paper, our approach is also using the deep learning techniques to train the models and classify methods into buggy or non-buggy. Our approach is different from the existing learning-based approaches in the following ways. First, like the rule-based approaches, the existing learning-based approaches do not consider the relations among paths across multiple methods. In our code representation learning step, we model the relations among paths from different methods using the dependencies of entities in the PDG and DFG, in addition to the AST nodes of a path. Second, our approach uses long paths of an AST to cover all of the AST nodes for representing local context, while other existing approaches often use part of method information to detect bugs, such as name-based identifier representation and frequent *n*-grams. Our results show that our approach can outperform all of the studied baselines.

6.2 Code Representation Learning

The recent success in machine learning has lead to strong interest in applying machine learning techniques, especially deep learning, to program language (PL) analysis and software engineering (SE) tasks, such as automated correction for syntax errors [Bhatia and Singh 2016], fuzz testing with probabilistic, generative models [Patra and Pradel 2016], program synthesis [Amodio et al. 2017], code clones [Li et al. 2017; Smith and Horwitz 2009; White et al. 2016], program classification and summarization [Allamanis et al. 2016; Mou et al. 2014], code similarity [Alon et al. 2018; Zhao and Huang 2018], probabilistic model for code [Bielik et al. 2016], and path-based code representation [Alon et al. 2018]. In the above PL and SE tasks, all of the above approaches learn code representations using different program properties. Although the learned code representations are not proposed for detecting bugs, they still very relevant to our study, as one important step of our approach is to learn bug detection specialized code representation. Different from the existing code representation learning approaches, we learn code representation using the AST, Program Dependency Graph and Data Flow Graph, and different attention-based neural networks. More

importantly, we incorporate the previous bug fixes into our code representation using an attention mechanism. Our results show that our code representation is more suitable for detecting bugs than the studied baselines, such as the baseline code representations using tokens and identifiers.

7 THREATS TO VALIDITY

We have identified the following threats to the validity:

Implementation of baselines. To compare with existing bug detection approaches, we have re-implemented a learning-based approach, *Bugram* [Wang et al. 2016a], since the *Bugram* code has been removed from the public repository. The source code of other baselines studied in our study is publicly available and we directly use their code in our experiments.

The *Bugram* paper reported a slightly higher precision and F-score than what we reported using our implementation of Bugram in this paper. One possible reason is that *Bugram* performs differently on different datasets and some implementation details are not mentioned in their paper, which makes our version of *Bugram* slightly different from the one in the original paper. However, we tried our best to build and tune the Bugram parameters on our dataset and this is the best effort we can make when the code is not publicly available. We tuned our approach and Bugram both on our dataset, which would make it fair for both of our approach and Bugram.

Applying all baselines on our dataset. Some baselines reported better results in their original papers than the ones we obtained in our paper. The main reason is that some baselines were not evaluated on Java code. Although we did not compare our tool with the baselines on their datasets, we compare all approaches on our collected dataset and tune them for the best results.

Collecting bug reports. During the bug report collection, we solely rely on the bug metadata that is manually created by the developers. We only download the bug reports with tags “bug" and “resolved or fixed". However, sometimes, a bug report marked as “bug" is not really a bug, but rather a code refactoring, which is commonly well-known problem in bug management. Due to the large amount of bug reports collected, we cannot verify all of them and make sure all of our data is correctly marked, which is common in large-scale data analysis. However, when evaluating our results, we also manually verify the top ranked 100 bugs to identify the true bugs to try our best to minimize this potential bias in our study.

Verifying true bugs in our qualitative analysis. Following prior studies [Bian et al. 2018; Gruska et al. 2010; Li and Zhou 2005; Livshits and Zimmermann 2005; Wang et al. 2016a], we manually check if the reported bugs in qualitative analysis are true bugs. Although this part of work is common in studies for bug detection, this process will bring bias to our results since the authors of this paper are not the developers of these projects. Sometimes we may misunderstand the code and come up with the wrong idea of a bug being a true bug.

Selection of programming languages. In this study, we only apply our approach on Java code. Thus, we cannot claim that our approach is generic for all programming languages. We choose Java code because Java is a widely used programming languages with many mature projects. However, the key drivers of our approach outperforming the baselines are general across programming languages: AST paths, PDG, DFG, and attention mechanism. Our methodology is general, no techniques/algorithms on the above extracted programming structures are tied to any programming languages. However, the results might be different for different languages. We have published our code and data in [Pro 2019] that can be learned to apply on other languages using the same trees/graphs: AST, PDG, and DFG.

8 CONCLUSION

In this paper, we propose a new deep learning based bug detection to improve the existing state-of-the-art detection approaches. The key ideas that enable our approach are (1) modeling and analyzing

the relations among paths of ASTs from different methods using the Program Dependency Graph (PDG) and Data Flow Graph (DFG); and (2) using weights and attention mechanism to emphasize previous buggy paths and differentiate them from non-buggy ones.

In our approach, we first propose an attention-based neural network to learn bug detection specialized code representation by incorporating previous bug fixes, paths of an AST within a method (i.e., local context), and program graphs modeling relations among ASTs of a whole project (i.e., global context). In the process of building local context code representation, we add weights to buggy paths and use three attention-based neural networks to learn contexts in the long paths on an AST. As a bug can involve multiple methods of a project, we model the relations among methods using the PDG and DFG as global context, and incorporate the global context into learned the local context code representation. Specifically, we encode PDG and DFG into low-dimensional vectors using the Node2Vec to learn global context path vectors. We combine the local and global path vectors into a unified path vector. Then, we append all path vectors of a method to obtain the method representation. Second, we use a CNN architecture to classify a given set of methods into buggy or non-buggy by analyzing the method representations.

We evaluate our approach against a set of state-of-the-art baselines in two settings: detecting bugs in an unseen project and detecting bugs in an unseen version of a project. Our empirical studies show that our approach can outperform all of the studied state-of-the-art approaches in both settings. Specifically, we can gain a relative improvement up to 168% in terms of F-score. Also our approach has a lower false positive rate than any baseline in both settings, which makes our approach more suitable for daily-practical use. Furthermore, compared with several types of existing code representations, our path-based code representation is more suitable for bug detection.

In the near future, we plan to evaluate our approach on different programming languages, e.g., Python and C, to gain more insights on the effectiveness of our approach on other PLs. Also, we plan to investigate other techniques to efficiently model graphs nodes and their surrounding structures.

ACKNOWLEDGMENTS

This material is based upon work partially supported by the US National Science Foundation (NSF) under Grant No. CCF-1723215, Grant No. CCF-1723432, Grant No. TWC-1723198, Grant No. CCF-1518897, and Grant No. CNS-1513263. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

2019. The GitHub Repository for This Study. (2019). https://github.com/OOPSLA-2019-BugDetection/OOPSLA-2019-BugDetection

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. *CoRR* abs/1602.03001 (2016). arXiv:1602.03001 http://arxiv.org/abs/1602.03001

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. *CoRR* abs/1803.09473 (2018). arXiv:1803.09473 http://arxiv.org/abs/1803.09473

Matthew Amodio, Swarat Chaudhuri, and Thomas W. Reps. 2017. Neural Attribute Machines for Program Generation. *CoRR* abs/1705.09231 (2017). arXiv:1705.09231 http://arxiv.org/abs/1705.09231

Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 1–8.

Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *CoRR* abs/1603.06129 (2016). arXiv:1603.06129 http://arxiv.org/abs/1603.06129

Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: Discovering Negative Association Rules from Code for Bug Detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA,

411–422. https://doi.org/10.1145/3236024.3236032

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. 2006. Improving Your Software Using Static Analysis to Find Bugs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 673–674. https://doi.org/10.1145/1176617.1176667

Yann Le Cun, Conrad C. Galland, and Geoffrey E. Hinton. 1989. Advances in Neural Information Processing Systems 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter GEMINI: Gradient Estimation Through Matrix Inversion After Noise Injection, 141–148. http://dl.acm.org/citation.cfm?id=89851.89868

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 57–72. https://doi.org/10.1145/502059.502041

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. *CoRR* abs/1607.00653 (2016). arXiv:1607.00653 http://arxiv.org/abs/1607.00653

Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 Projects: Lightweight Cross-project Anomaly Detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/1831708.1831723

Jordan Henkel, Shuvendu Lahiri, Ben Liblit, and Thomas W. Reps. 2018. Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. *CoRR* abs/1803.06686 (2018). arXiv:1803.06686 http://arxiv.org/abs/1803.06686

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

David Hovemeyer and William Pugh. 2007. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 9–14. https://doi.org/10.1145/1251535.1251537

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. *SIGPLAN Not.* 47, 6 (June 2012), 77–88. https://doi.org/10.1145/2345156.2254075

Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 194–206.

Hyeji Kim, Yihan Jiang, Sreeram Kannan, Sewoong Oh, and Pramod Viswanath. 2018. Deepcode: Feedback Codes via Deep Learning. *CoRR* abs/1807.00801 (2018). arXiv:1807.00801 http://arxiv.org/abs/1807.00801

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. https://doi.org/10.1109/ICSME.2017.46

Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 306–315. https://doi.org/10.1145/1095430.1081755

Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 333–344. https://doi.org/10.1145/2884781.2884870

Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 296–305. https://doi.org/10.1145/1095430.1081754

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013b. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013 (NIPS'13)*. 3111–3119.

Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes using Historic Databases.. In *icsm*. 120–130.

Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR* abs/1409.5718 (2014). arXiv:1409.5718 http://arxiv.org/abs/1409.5718

Jaechang Nam and Sunghun Kim. 2015. Heterogeneous Defect Prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 508–519. https://doi.org/10.1145/2786805.2786814

Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009a. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (FASE'09)*. Springer-Verlag, 440–455.

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009b. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. *SIGPLAN Not.* 50, 6 (June 2015), 369–378. https://doi.org/10.1145/2813885.2737966

Jibesh Patra and Michael Pradel. 2016. Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data.

Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection. *CoRR* abs/1805.11683 (2018). arXiv:1805.11683 http://arxiv.org/abs/1805.11683

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

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 155–165.

Randy Smith and Susan Horwitz. 2009. Detecting and Measuring Similarity in Code Clones.

Soot. [n. d.]. Soot Introduction. https://sable.github.io/soot/. ([n. d.]). Last Accessed July 11, 2019.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/3196398.3196431

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

WALA. [n. d.]. WALA Documentation. http://wala.sourceforge.net/wiki/index.php/Main_Page. ([n. d.]). Last Accessed July 11, 2019.

Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016a. Bugram: Bug Detection with N-gram Language Models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 708–719. https://doi.org/10.1145/2970276.2970341

Song Wang, Taiyue Liu, and Lin Tan. 2016b. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 297–308. https://doi.org/10.1145/2884781.2884804

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1287624.1287632

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. 2015. ABCNN: Attention-Based Convolutional Neural Network for Modeling Sentence Pairs. *CoRR* abs/1512.05193 (2015). arXiv:1512.05193 http://arxiv.org/abs/1512.05193

Edward Yourdon. 1975. Structured Programming and Structured Design As Art Forms. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition (AFIPS '75)*. ACM, New York, NY, USA, 277–277. https://doi.org/10.1145/1499949.1499997

Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068