

Leveraging Transformer and Graph Neural Networks for Variable Misuse Detection

Vitaly Romanov, Gcinizwe Dlamini^a, Aidar Valeev and Vladimir Ivanov^b

Faculty of Computer Science and Engineering, Innopolis University, Innopolis, Russia

Keywords: Graph Neural Network, CodeBERT, Variable Misuse Detection.

Abstract: Understanding source code is a central part of finding and fixing software defects in software development. In many cases software defects caused by an incorrect usage of variables in program code. Over the years researchers have developed data-driven approaches to detect variable misuse. Most of modern existing approaches are based on the transformer architecture, trained on millions of buggy and correct code snippets to learn the task of variable detection. In this paper, we evaluate an alternative, a graph neural network (GNN) architectures, for variable misuse detection. Popular benchmark dataset, which is a collection functions written in Python programming language, is used to train the models presented in this paper. We compare the GNN models with the transformer-based model called CodeBERT.

1 INTRODUCTION

In the context of this work, to find misused variables in source code is to detect incorrect occurrences of variables. Detecting such misused variables can prevent bugs and vulnerabilities in programs, which are difficult to recognize, because typically misused variables do not lead to compiling errors. Statistical analysis and machine learning approaches are increasingly used, including both traditional natural language processing approaches (Vasic et al., 2019) and approaches that use graph-based representation of source code (Allamanis et al., 2017). In general, the essence of the problem lies in the detection of misused variables, as well as the recommendation of the correct candidate, which might be one of the variables already present in the body of a function.

In this paper, we propose and evaluate a graph neural network (GNN) architecture for the problem of variable misuse detection. The problem includes two subtasks: localization of an incorrectly used variable and its repair by identifying a correct variable in the scope. When source code is represented as a graph, both subtasks can be formulated as node classification problems.


Recently, vector representations for source code elements were trained using convolutional networks


on graphs (GCNs). One of the primary structures used for this is abstract syntax tree. To build a program graph, in addition to the syntax tree, external information can be used. For example, the place in the program where the last use of the variable occurred.

Despite the conceptual advantage of representing source code as a graph, existing methods for training machine learning models on graphs have few disadvantages. Most existing models are based on the message passing mechanism. As a consequence, a large amount of message passing may be required to model links between far-flung nodes in a graph. In (Helleendoorn et al., 2019), it is proposed to use a hybrid approach, which simultaneously uses the representation of the source code as a sequence and as a graph. This approach enables using the strengths of various neuro-architectures in a single model.

The goal of this paper is to explore the capabilities of Graph Neural Networks for solving the problem of variable misuse detection. We compare performance of different GNN architectures and identify challenges that emerge when using GNNs for source code analysis. Specifically, we observed that the complexity of the input graph negatively affect the quality of variable misuse detection.

In our tests of GNN models on the task of variable misuse detection, we observed that pre-trained models such as CodeBERT achieve far better performance than GNN models trained from scratch. However, GNN models used in our experiments use far

^a  <https://orcid.org/0000-0002-4578-5011>

^b  <https://orcid.org/0000-0003-3289-8188>

fewer parameters, which make the conclusion about the results more problematic.

The structure of this paper is as follows: Section 2 outlines the related work together with background on graph neural networks and variable misuse. In Section 3, the proposed model is presented in detail, while Section 4 presents the obtained results followed by a discussion. The paper is concluded in Section 5 with possible future directions.

2 BACKGROUND

In this section, we present a background on the methods used in this paper.

2.1 Graph Neural Network

A Graph Neural Network (GNN) is a machine learning model specially designed to learn and extract knowledge from graph-structured data (Gori et al., 2005). From the graph theory, a graph $G = (V, E)$ is made of nodes $V = v_1, \dots, v_n$ and set of edges $E = E_1, \dots, E_k$ connecting the nodes. The general idea behind GNNs is representing each node in a graph with a vector, whereby the vector represents the node's neighbors and the relationship between the intermediate nodes.

Over the years, there have been many variations in the implementations of graph-structured format for training GNNs. One of the most popular realizations of GNNs emerged from quantum chemistry presented by Gilmer et al. (Gilmer et al., 2017) called the message-passing technique. In the message-passing technique, the state of a node (vector representation) is computed by aggregating the signals for the neighboring nodes. In each iteration a node v representation as :

$$s_v^{(i+1)} = \phi \left(\{s_u^{(i)}\}_{u \in N(v)} \right) \quad (1)$$

where $N(v)$ is the set of all the neighbors that are connected to v , $\phi(\cdot)$ is a non-linear function that performs the aggregation of information, $s_v^{(i)}$ denotes the state of node v at step i .

A relational graph convolutional network (RGCN), has been used in recent papers (Liu et al., 2021) and (Ling et al., 2021). Initially, all nodes are initialized using vector representations of nodes. Then several iterations of message transmission are performed. Aggregation of messages is performed after each iteration by averaging vector representations received from neighbors.

2.2 Transformer Architecture

Transformer (Vaswani et al., 2017) has proved its efficiency in many fields, including Natural Language Processing and Computer Vision. Most pre-trained transformer follow the Encoder-Decoder architecture. A single encoder layer consists of two blocks: a self-attention block and a feed-forward network. Multi-Head Dot-product Self-Attention block is defined as:

$$SelfAttention(X) = Softmax \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2)$$

where Q , K , and V are linear projections of X , d_k is the dimensionality of K . A single decoder layer is similar but extended with an attention-to-encoder block, which has encoder outputs as V in the self-attention formula.

A transformer is designed for sequence-to-sequence tasks, therefore BERT (Devlin et al., 2019) appeared as a universal solution to many Natural Language Processing tasks.

CodeBERT (Feng et al., 2020) is a pretrained BERT model for programming and natural languages. CodeBERT was pretrained on the dataset from GitHub repositories in six programming languages. This dataset consists of bimodal (2.1 million samples) and unimodal (6.5 million samples), where bimodal means pairs of functions and corresponding summaries, while unimodal means solely functions.

Researchers over the years have proposed different Transformer based approaches to detect variable misuse. Vasic et al. (Vasic et al., 2019) in addition to detecting if a variable misuse using LSTM coupled with attention, proposed an approach that pinpoints the location of misuse and generates a repair. The researchers used two datasets as benchmarks: ETH-Py150 and MSR-VarMisuse (25 C# GitHub projects). To validate the LSTM-based approach, the researchers achieved an accuracy of 82.4% on variable misuse, 71% on localization, and 65.7% on localization+repair task. To further validate the proposed approach, Vasic et al. (Vasic et al., 2019) evaluated their model on realistic scenarios where they collected a dataset from multiple software projects in an industrial setting. The accuracy of the model on realistic scenarios is 66.7%, 21.9% on localization and 15.8% on localization+repair.

In the same spirit as Vasic et al. (Vasic et al., 2019), (Chirkova, 2020) two years later proposed an approach for generating dynamic embeddings aimed at improving the performance of the recurrent neural network, in code completion and bug-fixing tasks. The researchers focused their research on two popular dynamically typed programming languages, namely JavaScript and Python. The dataset used for training

and testing are Python150k (Raychev et al., 2016a) and JavaScript150k (Raychev et al., 2016b)

3 METHODOLOGY

In this section, we present our methodology and the implementation details. Figure 1 presents an overview of our pipeline.

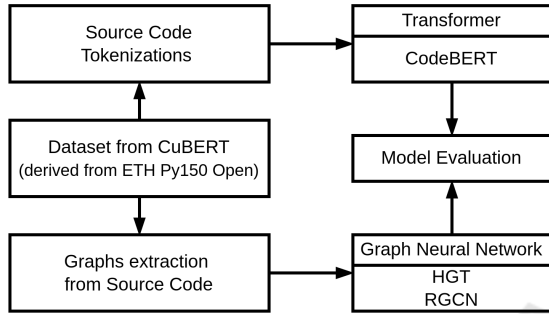


Figure 1: The Pipeline represents two alternative ways for evaluating the performance: Transformer-based and GNN-based.

3.1 Dataset Creation

The dataset used to train the variable misuse detection model is based on the benchmark dataset prepared as part of the CuBERT paper (Kanade et al., 2020). The part of the dataset related to variable misuse localization and repair is used. However, the data representation format is not suitable for converting the source code into a graph, since the source code is given as a list of CuBERT tokens. In order to adapt the dataset for the current study, the following procedure is used.

The Variable-misuse classification part of the dataset is used as the basis for the dataset, which does not contain information about the location of the error. Only information about whether an error is present in a given function, as well as which variable names are used incorrectly. In order to prepare the data suitable for transformation into a graph, we exploit the fact that the reference dataset contains the example of the correct function and the example of the same function with the error. Thus, it is possible to reconstruct in which line the incorrectly used variable is located. The Variable-misuse classification part of the reference dataset processed in this way contains examples of functions that can be unambiguously matched to functions in the Variable-misuse localization and repair part of the reference dataset. This fact allows us to compare the results obtained on the dataset prepared in this work with the results published for the Variable-misuse localization and repair problem.

The transformation of source code into a graph is performed as follows. The following requirements are made to a graph representation of the source code:

1. the variables with the same name occurring in the body of one function shall be interpreted as the same variable;
2. the expressions defined in the body of if conditional operator, for and while loops, try exception handling block shall be linked to the above-mentioned operators (for example, an expression in if operator block shall be linked to the node corresponding to this operator);
3. the order of expressions execution shall be reflected in the graph;
4. links with imported modules, called functions, and inherited classes must be unambiguously resolved;
5. values of constants in the source code (numbers and strings) must not be present in the graph;
6. function and variable names must be tokenized.

The first step is the extraction of global relationships, which is done with Sourcetrail. During the indexing process, Sourcetrail creates a database of global relationships. Examples of such relationships are relations to functions called in code, imported modules, and inheritance relations. The $g_{global} = (N_{global}, E_{global})$ graph represents the set of global nodes and the relationships between them. The Sourcetrail utility allows to extract dependencies even when importing from third-party packages. As a result, information is collected for frequently used libraries about exactly how they are used. In addition, Sourcetrail stores the correspondence between the source code and the nodes in the graph, which allows later to use the combined representation of source code as a sequence of tokens and as a graph to solve the type classification problem.

Further, source code in the codebase is processed at the individual file level. The ast module (Python 3.8) is used to extract the syntax tree of the program. Next, a local source code graph $g_{local} = (N_{local}, E_{local})$ is generated for an individual file. The step of combining variable names transforms the syntax tree of the program into a graph. Variables with the same name inside the function body are combined into one node. All references to the function name inside the file are also represented by one node in the graph. Then auxiliary edges are added. For expressions defined in the body of the conditional if statement, for and while loops, and the try exception handling block, links are added to the statements mentioned above (for example, for an expression in the if statement

block, a link is added to the node corresponding to this statement). Additional edges *next* and *prev* are used to show the order of program execution in the graph.

The last step of creating a local graph is the tokenization of names. In this step, nodes and edges of subword type are added to the graph, which represents names tokens. The nodes representing tokens are common to all files in the code base. Without tokenization, the number of unique names grows due to neologisms as new code is added to the code base (Al-lamanis et al., 2017). One of the most popular tools for tokenization is sentencepiece (Kudo and Richardson, 2018). It is based on compression algorithms, finds the most frequent substring in the code base, and uses them as tokens.

At the last stage of file processing, the global nodes are mapped in the code base. The edges of global mention type are added to the local graph, linking the nodes of the local graph with their corresponding nodes of the global graph. These edges can be represented by the graph

$$g_{\text{global mention}} = (N_{\text{local}} \cup N_{\text{global}}, E_{\text{global mention}}) \quad (3)$$

The result of matching is graph :

$$g = g_{\text{local}} \cup g_{\text{global}} \cup g_{\text{global mention}} \quad (4)$$

After that, the graph g for an individual file is merged with the general graph G , which combines all processed packets.

3.2 Graph and Transformer Models

For conducting experiments, two models of graph neural networks were tested: (1) relational graph neural network (RGCN (Schlichtkrull et al., 2018)) and (2) heterogeneous graph transformer (HGT (Hu et al., 2020)). We use HGT for comparison for two reasons. First, this model is newer than RGCN and showed better performance on some tasks. Second, since transformers often show superior performance, it is only natural to consider a GNN model that is based on the transformer architecture.

During experiments with function-level variable misuse detection, we test the pre-trained CodeBERT model from the transformer library.

GNNs, such as RGCN and HGT, work in the following way. At the initial stage, the nodes corresponding to the source code tokens are initialized with vector representations defined for these tokens. For the remaining nodes of the graph, initialization is performed using vector representations of node types. Then several iterations of message transmission are performed. Aggregation of messages is performed

after each iteration by averaging vector representations of neighboring nodes. In HGT, aggregation uses transformer with self-attention. In RGCN, summation is used as the aggregation function.

3.2.1 Variable Misuse Detection as Node Classification

The detection of variable misuse can be performed at a lower level of abstraction, whereby each variable in a given method is classified as containing a misused or not. To predict if a variable is misused in the context of the given method (scope), it is important to capture the context and, based on the context, decide whether a variable is misused or not. The main task is to capture the context and assign a label to each of the variables in the scope.

To detect whether a variable is misused or not, we formulated the task as a node classification problem. Each program is represented by a graph. The nodes in a graph represent variables and the edges represent their relationships. Each node that represents a variable can be assigned a binary label based on whether this variable was misused or not.

3.2.2 Function-Level Variable Misuse as Subgraph Classification

The detection of variable misuse can be performed at a higher level of abstraction, where methods are classified as containing a variable misuse or not. To predict whether a method contains a misused variable, the method is represented as a subgraph, which is a part of a program.

To represent the subgraph as an embedding for variable misuse prediction, the message passing is performed only among the subset nodes in the subgraph. The propagation of the subgraph information and calculation of the vector representation of a function embedding is achieved by applying pooling after the message passes between the nodes.

Pooling approaches are commonly used in convolutional neural networks when dealing with image data. For graph-structured data, features from a set of nodes of undefined size should be pooled into a vector of fixed dimensionality. In our research, the following variants of such functions were considered :

- Calculation of the vector representation of the function by averaging the vector representations of nodes (**Average Pooling**);
- Calculation of the vector representation of the function by weighted averaging, weights for averaging are calculated using the Attention Pooling method (**Muti-head attention pooling**);

- Calculation of the vector representation of the function by weighted averaging of the first k nodes with the largest weights, the weights for averaging are calculated using the attention method (**U-Net Pooling** (Gao and Ji, 2019)).

3.3 Models Training and Evaluation

We train all GNN-based model which is a black box analysis learning task and learn the relationship of the variable in a given scope to predict if there exists a variable misuse. The implementation is in python using PyTorch deep learning framework. For the purpose of conducting experiments and searching for optimal parameters, we varied the three parameters (batch size, node embedding size, and number of layers in the neural network).

To evaluate the trained models, we used the accuracy metric. To observe and analyze the model performance we used different configurations whereby for GNN different pooling methods, number of layers, size of the training dataset, and the dimension of vector representation we used.

4 RESULTS AND DISCUSSIONS

In this section, we present the achieved experimental results and the discussion of the results.

We conducted pilot experiments on the 10% sample of the full dataset. Originally we focused more on the node-level variable misuse detection. The results for there experiments are shown in Table 1. Experiments on the full dataset included only variable misuse detection on the function level because these results were easily comparable with similar metrics produced using CodeBERT classifier. The results for these experiments are shown in Table 2. In both experiments, the reported accuracy scores are the best accuracy on testing (as Test best) and training (Train best) set during the model training.

The results of the first series of experiments using the 10% portion of the dataset, trained on the task of node-level variable misuse detection, show that the model does not experience significant overfit. The accuracy of misuse detection increases slightly with the increase of GNN dimensionality. The number of layers does not affect the results.

The method level variable misuse detection was conducted on the full dataset. In this set of experiments, we tested several GNN models: Hierarchical Graph Transformer (HGT), Relational Graph Convolutional Network (RGCN) (Schlichtkrull et al., 2018), and CodeBERT. CodeBERT is a pre-trained model,

Table 1: Variable misuse detection at the node level with GNN and 10% of dataset. The base rate of misused variables is 25%.

Dimension of vector representations	Number of layers	Accuracy	
		Test best	Train best
100	5	0.836	0.87
100	3	0.83	0.88
100	5	0.83	0.84
100	3	0.837	0.88
50	5	0.836	0.85
50	3	0.82	0.82
50	5	0.82	0.84
50	3	0.82	0.86
30	5	0.82	0.84
30	3	0.82	0.85
30	5	0.81	0.82
30	3	0.82	0.84

that was trained using a large collection of source code. GNN models are trained from scratch. Among GNN models, RGCN achieves the best performance when used together with attention pooling. CodeBERT does not perform well before fine-tuning. After fine-tuning, CodeBERT achieves far smaller error rate than any of GNN models.

From Table 2 the comparison of different models with different architectures is presented. The fine-tuned CodeBERT model outperforms all the other models with a significant margin. This could be because of the amount of information embedded in the vector size of 768 compared to the one of GNN (max 300). Incorporating attention pooling improved the performance of RGCN model since with 3 layers it achieved better results, this could be explained by the benefits and power that comes with the attention mechanism. On the other hand, U-net pooling which has the mechanism slightly similar to attention and designed for graph structured data seems to fall behind attention pooling.

Based on our conducted experiments, we are inclined to conclude that the incorporation of transformers (i.e. as pooling mechanism) is quite promising in the task of variable misuse detection. There is some still more room for improvements in the use of GNNs for tasks such as variable misuse. When assessing the quality of classification models of incorrectly used variables, it was revealed that graph neural networks cannot solve the task when using the source code graph format and graph neural network architectures specified in this paper.

5 CONCLUSION

In this paper, we present an analysis of the challenges faced by graph neural networks. For testing models

Table 2: Models comparison in variable misuse detection task evaluated at the method level. The base rate of functions with misuse is exactly 50%.

Model	Dimension of vector representations	Number of layers	Pooling Method	Number of training epochs	Accuracy	
					Final Test	Test best
RGCN	100	3	Average Pooling	10	72.9	73.53
HGT	100	3	Average Pooling	10	66.12	66.44
HGT	300	5	Average Pooling	20	70.96	71.01
RGCN	300	5	Average Pooling	20	77.36	77.36
RGCN	300	5	U-net Pooling	20	76.83	76.83
RGCN	300	3	Attention Pooling	20	77.88	78.36
RGCN	300	3	U-Net Pooling	20	74.95	76
CodeBERT with fine-tuning	768	8	-	20	95.97	95.97
CodeBERT without fine-tuning	768	8	-	20	58.55	58.55

of graph neural networks, two variations of the problem of detecting misused variables are considered: (1) classification of individual variables and (2) classification of functions (or methods) for the presence of incorrect variables in their bodies. The classification accuracy of functions for the presence of misused variables was measured. Graph neural network models achieve significantly lower classification accuracy than the CodeBERT model. For the future research, we plan investigating the training process of graph neural networks in order to improve the performance by focusing on more recent graph format mainly HGT with more optimized training parameters. In addition to tuning HGT architecture for better performance in variable misuse detection, we are aimed at minimizing the problem of input graph complexity for HGT model.

ACKNOWLEDGEMENTS

The study was funded by a Russian Science Foundation grant number 22-21-00493 <https://rscf.ru/en/project/22-21-00493/>.

REFERENCES

- Allamanis, M., Brockschmidt, M., and Khademi, M. (2017). Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.
- Chirkova, N. (2020). On the embeddings of variables in recurrent neural networks for source code. *arXiv preprint arXiv:2010.12693*.
- Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T., editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis,*

MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pages 4171–4186. Association for Computational Linguistics.

- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. In Cohn, T., He, Y., and Liu, Y., editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.
- Gao, H. and Ji, S. (2019). Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR.
- Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR.
- Gori, M., Monfardini, G., and Scarselli, F. (2005). A new model for learning in graph domains. In *Proceedings. 2005 IEEE international joint conference on neural networks*, volume 2, pages 729–734.
- Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. (2019). Global relational models of source code. In *International conference on learning representations*.
- Hu, Z., Dong, Y., Wang, K., and Sun, Y. (2020). Heterogeneous graph transformer. In *Proceedings of the web conference 2020*, pages 2704–2710.
- Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.
- Kudo, T. and Richardson, J. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.
- Ling, X., Wu, L., Wang, S., Pan, G., Ma, T., Xu, F., Liu, A. X., Wu, C., and Ji, S. (2021). Deep graph matching and searching for semantic code retrieval. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(5):1–21.

- Liu, L., Nguyen, H., Karypis, G., and Sengamedu, S. (2021). Universal representation for code. In *Advances in Knowledge Discovery and Data Mining: 25th Pacific-Asia Conference, PAKDD 2021, Virtual Event, May 11–14, 2021, Proceedings, Part III*, pages 16–28. Springer.
- Raychev, V., Bielik, P., and Vechev, M. (2016a). Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.
- Raychev, V., Bielik, P., Vechev, M., and Krause, A. (2016b). Learning programs from noisy data. *ACM Sigplan Notices*, 51(1):761–774.
- Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. (2018). Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*, pages 593–607. Springer.
- Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. (2019). Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*, pages 5998–6008.

