Attempts on applying graph neural network to cross-language code-clone detection

Yao Xuyang, Shigeru Chiba

クロスプラットフォームアプリケーションやソフトウェアの移植が増加しているため，異なるプログラミング言語の間での機能の重複は回避できなくなってきている．これらのようにプログラミング言語が異なる類似したコードの繰り返しは言語間コードクローンとして知られている．複数のプログラミング言語の間での言語間コードクローンの検出はクロスプラットフォームアプリケーションのバージョンコントロールやバグ修正のために非常に重要である．しかし言語間コードクローンはふつう構文的にはあまり似ていない．このことが検出を難しくしている．この問題領域ではすでに多くの研究が行われている．その中のいくつかはソースコードの構文的構造，抽象構文木 (Abstract Syntax Tree, AST) を用いる．これは AST が，プログラミング言語が異なるにも関わらずコードクローンの類似性を共有すると思われるからである．一方で，同一言語のコードクローン検出にグラフの構造情報を抽出できる革新的なコンセプトであるグラフニューラルネットワーク (Graph Neural Network, GNN) を利用する研究も進められており，多くの結果を残している．我々の研究では GNN モデルを言語間コードクローン検出に適用しようと試みる．本論文ではいくつかのモデルの設定と関連する実験と結果，そして今後の構想を示す．

Due to the increase in cross-platform applications and the transplantation of software, repetitions of functionality in different programming languages become unavoidable. These repetitions of similar code in different programming languages are known as cross-language code-clones. Detecting cross-language code-clones implemented in different programming languages is vital for tasks like version control and bug fixing for cross-platform applications. However, cross-language code-clones are usually syntactically different to each other which make them difficult to be detected. Many studies have already been conducted in this area, some of them work on abstract syntax tree (AST), a syntactical structure of source code, which may share some similarity between code-clones despite of the difference in programming language. Meanwhile, there has been researches conducted on monolingual code-clone detection utilizing a novel concept called Graph Neural Network (GNN), which is capable of extracting structural information from graphs and achieved significant progress. In this work, we tried to apply a GNN model to the task of cross-language code-clone detection. This paper will be about several model settings and corresponding experiments and results, finishing with some future thoughts yet to be finished or conducted.

1 Introduction

Code-clones refers to pairs of code fragments that share some level of syntactical or semantical similarity. Undiscovered code-clones can cause potential problems in bug fixing, version maintenance or even runtime efficiency. With the development of cross-platform applications and increase of package transplanting among programming languages, code-clones are occurring more and more frequently among different languages. Thus there has been rising demand on efficiently detecting cross-language code-clones.

However, the difference of syntax and keywords between programming languages greatly stressed the difficulty of syntactical detection of cross-language code-clones, which makes the monolingual code-clone detection approaches no longer feasible

* グラフニューラルネットワークを言語間コードクローン検出に適用する試み

姚 旭楊, 千葉 滋, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, University of Tokyo .

on the cross-language task. Currently, in order to detect semantical similarity between cross-language code pairs, many recent works has been utilizing intermediates such as Control Flow Graph (CFG) and Abstract Syntax Tree (AST) which contains code semantics in their structure. The literature [4] compared the similarity of source code by directly comparing a modified AST; the literature [5] generated vector representation for ASTs with a recurrent neural network (RNN) model with long short-term memory (LSTM) to compute their similarity.

On the other hand, a new class of deep learning model, Graph Neural Network (GNN), has been proposed. GNN is a class of connectionist models that capture the dependence of graphs via message passing between the nodes of graphs. Due to its capability of process non-linear graph structures, we believe GNN is also feasible for extracting information and comparing similarity of ASTs. Although plenty of GNN models has been proposed, as far as we know currently there is no precedent on applying them to cross-language code-clone detection.

In this paper, we present our attempt on proposing a GNN based model for detecting cross-language code-clones. Our model is expected to generate embeddings from ASTs, which could be used to compute the similarity between corresponding code fragments.We build the model by implementing and modifying existing GNN models, and training them with specially designed data set. Our work mainly contributes as follows:

- To the best of our knowledge, we are the first to attempt applying GNN models on cross-language code-clone detection.
- We tried to automate the code embedding stage by data augmentation, without the need of a huge well-labeled data set.
- We studied the possibility of graph neural network models detecting similarity between heterogeneous graphs.

The remainder of this paper is arranged as follow. In section 2 we describe our motivation of choosing to apply GNN models on ASTs. Then in section 3 we present the implementation of our model. The experiments and results are provided in section 4, following with comparison with a close work [7] section 5. Finally in section 6, we will give a conclusion to this paper and talk about our plan for next step.

2 Extracting semantic information from ASTs

In this work, we aim to detect cross-language code-clones by extracting semantic information from abstract syntax tree (AST) with graph neural network (GNN) models. Clone detection has been a long studied topic and there has been promising results on syntactical similar code-clone detection. However, it remains an unsolved problem to efficiently detect code-clones that are similar in semantics but different in syntax, especially when they are from different languages. Based on existing GNN models, we propose to train an embedding generator that projects code fragments into a latent space according to their functionality. Thus similarity of code pairs can be computed with their embeddings, even if they are from different languages.

Code-clones are commonly classified into four types. While Type I-III clones share much similarity on their source code, Type IV and weak Type III code-clones can be quite different on the text. Due to the fundamental difference in syntax between programming languages, all cross-language code-clones are classified into Type IV. In order to detect such clones, semantical information usually matters more than syntactical details. Thus it has become a common approach to detect Type IV, especially cross-language code-clones, by utilizing Abstract Syntax Tree (AST).

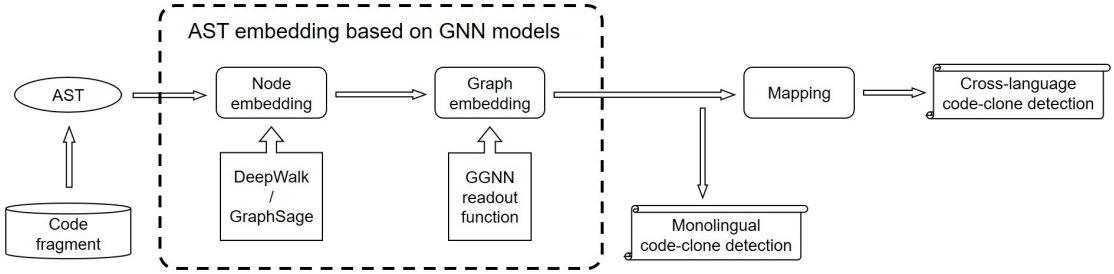


Figure 1 Overview of our proposed cross-language code-clone model. In this work, we implement the node embedding and graph embedding stage by localizing existing GNN models, and focus on the design of data set and the mapping stage to overcome the structural difference between ASTs from different programming languages.

AST is a tree representation of the abstract syntactic structure of source code written in programming language. By "abstracting" the "syntax", ASTs ignore most of the text level details while preserving the structure of the source code. Typically, indentation in Python and curled brackets in Java which preserves the block information will only result in the parent-children relationship in an AST. Thus it is a common case where different code fragment may share roughly similar ASTs.

There are more advantages for representing code with AST. In an AST, each node is linked with its neighbors in a clear hierarchical structure, which may further indicate the general functionality of the corresponding block of code. In other words, a small neighborhood of nodes in an AST could possibly present the information contained in a long sequence of tokens. Moreover, due to the hierarchical structure of AST, it's easy to extract information for different granularity of the source code, simply by starting from different node.

Due to the non-linear tree structure of AST, it cannot be directly processed by common linear approaches. There have been different attempts on computing the similarity between ASTs. [6] linearized the AST with traverse of a certain order so that the resulted linear sequence can be compared with Smith-Waterman local sequence alignment algorithm. [7] flattened the tree structure in the depth order so that it can be fed in to a recurrent neural network (RNN) with long short-term memory (LSTM).

On the other hand, graph neural network is a class of model that take graphs as direct inputs and extract their structural information, in the case of code fragment, semantic information. For a typical GNN model, it gathers information of each node and its neighbors as the final node embedding output. In the case of AST, such neighborhood information of each node is expected to represent a partial functionality of the source code. Finally, we believe aggregating the node embeddings would generate a representation that acts as a 'summary' of the source code functionality, which can be utilized for code-clone detection.

3 AST embedding models

In this section we will describe the models we proposed for the task of cross-language code-clone detection. Our general idea is to generate embeddings for code fragments from different languages that follows same local distribution. Then we project the embeddings of one programming language to the other with a mapping function to blur the language-wise difference, so that similar code from

different programming languages could have close embeddings.

Our embedding model consists of node embedding and graph embedding. For the node embedding stage, we implemented the existing GNN models DeepWalk [6] and GraphSage [1] for comparison. Since the DeepWalk model is originally designed for single graph node embedding generation, we specially modified the structure of data to fit the input for the DeepWalk model. For the graph embedding stage we modified the gated readout function proposed in the GGNN model [3]. Then we trained our model with a data set generated with special augmentation rule. Finally, since it's an early attempt, we simply applied a MLP as the mapping function.

In the remainder of this section, we first describe the setting of the node embedding and graph embedding model. Then we introduce our idea for data augmentation and embedding mapping.

3.1 Node embedding model

In this stage, we re-implemented two existing GNN models [1] [6] for node embedding. The aim of this stage is to capture the structural information from the neighborhood of each model, which could potentially indicate the local functionality. The results generated by the two models are used for the next stage respectively, for comparison.

3.1.1 DeepWalk

DeepWalk [6] is a commonly used approach for generating latent representation of vertices in a graph. For each training step, starting from an arbitrary node, the model randomly picks a route of certain hops according to some pre-designed transition rule. This route is called a random walk, and it's treated as the equivalent of a sentence. Then the generated sentences are considered as a special language, and processed by a neural language model to capture neighborhood similarity and community membership.

However, DeepWalk is a embedding model for single graph, and it lacks generalization ability across graphs, which means for the same node label in different graphs, it's very likely that DeepWalk will generate different embedding. So we first treated the whole data set as a single disconnected graph. To make the training process more intuitive, we further combined all the nodes with same label into a single node. And we assigned the number of occurrences of each edge as its weight, thus during the random walk stage, the probability of choosing each neighbor node will be proportional to the weight of the edge. In order to generate similar embedding for different different programming languages, in our case Python and Java, we further merged similar nodes, e.g. IF node in Python and Java.

As our expectation, a collection of information in a certain neighborhood of AST could represent local functionality. In this model setting, generated node embeddings are shared across all ASTs after the training phase. For the embedding of each node, it generalizes all possible situation of functionality with occurrence of the node, instead of representing the specific local functionality of the node in a certain AST. In such case, the proposed model blurs the differences in usage of the same node. And since node embedding network is learned in an unsupervised way and is separated from graph embedding network, this model requires relevantly less computational cost.

3.1.2 GraphSage

In order to capture more code-dependent local functionality information, we applied the GraphSage[13] model to generate node embedding. Instead of extracting substructure from the graph, the GraphSage model directly learns a function that generates embeddings by sampling and aggregating features from a node's local neighbor-

hood. With the constraint of the same generation function, GraphSage model is capable of generating stable (slightly different according to neighborhood structure) for same node label across different graphs.

However, GraphSage is a supervised learning model which requires labels (typically classification) for each node, which is not required in the graph classification step of our model. Instead of training the node embedding before aggregating them for graph embedding, we directly aggregate the untrained node embeddings and train the node embedding network and graph embedding network together by providing graph-wise labels.

This model focus more on the specific local structure of each individual AST, and the node label does not take significant part in the output. So we tried both separated and merged node labels in generating embeddings for different programming languages.

3. 2 Graph embedding model

With the generated node embedding as local information, we further apply an aggregation function to compute graph embedding as a 'summary' of the corresponding code fragment. In this work, we used the aggregation model proposed in the literature [3]:

$$h_G = MLP_G(\sum_{i \in V} \delta(MLP_{gate}(h_i)) \odot MLP(h_i)) \tag{1}$$

Intuitively, this model works as an aggregation function with attention mechanism. First the node embedding is processed by a simple MLP. At the same time, it go through a Gate network MLP_{gate} which outputs a importance weight for each dimension of the processed node embedding. Finally all the weighted node embedding is aggregated (sum, average, max pooling, etc.) and output through a final aggregation network MLP_G.

By the observation that in an AST, the nodes closer to the root node usually have a closer relationship to the main function of the code, we also trained a weight parameter for each layer of the AST to utilize the structural information:

$$h_G = MLP_G(\sum_{i \in V} \delta(MLP_{gate}(h_i)) \odot MLP(h_i) \times layer(i)) \tag{2}$$

where $layer(i)$ refers to the weight parameter of the layer of node i.

3. 3 Learning code space

Unlike common classification tasks, code-clone detection task is a pair-wise classification task where there is no certain label for each single code. Instead of mapping the input code fragments into a finite set of discrete class, our model intend to project them into an infinite continuous latent space, where the distance between each pair of embedding indicates the similarity of the corresponding code fragments. This can be achieved with pair-wise labels using a deep similarity learning method called TripletMarginLoss[15]:

$$L_{Triplet}(E, E^+, E^-) = max(d(E, E^-) - d(E, E^+), m) \tag{3}$$

Given an arbitrary output embedding E, and embeddings from a positive sample E^+ and a negative sample E^-, the loss function encourage the distance between negative pairs $d(E, E^-)$ to be greater than positive pairs $d(E, E^+)$ by a least a margin m. However, the clone pairs given by common code clone data sets are usually too similar, and the positive samples are limited to the clone pairs. An ideal data set should contain a relative distance 'label' for arbitrary pair of code, which is not available.

In this work, we propose to inject positive and negative samples to the original data set by utilizing data augmentation. Currently the injected samples are generated by changing node types and

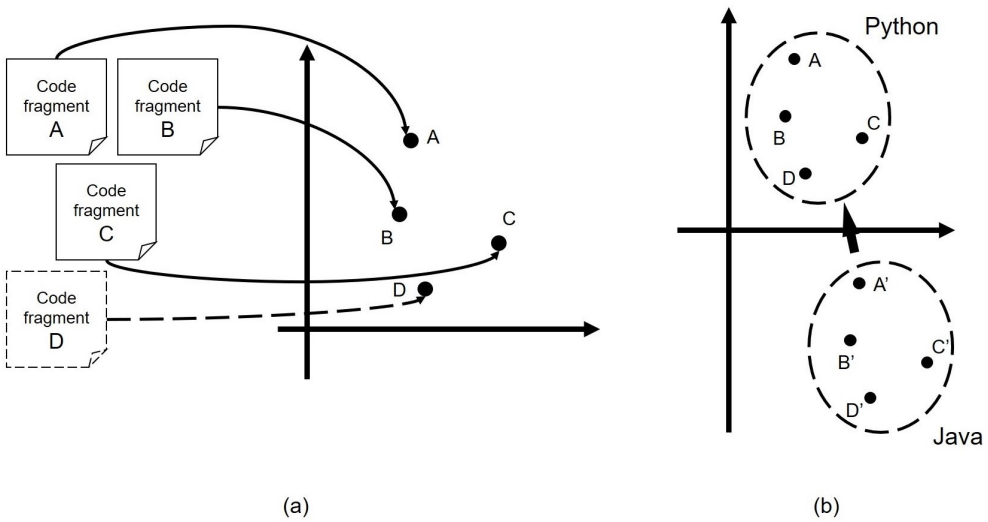


Figure 2 General idea for latent space learning and mapping. In figure (a), solid lines represents the training phase, closer distance between embeddings indicates higher similarity between corresponding code fragments. Dotted line shows when the model tries to embed a new code fragment unseen in the data set. Figure (b) shows the mapping stage designed for cross-language code-clone detection. By mapping embeddings from one language cluster to another, we implicitly remove the language-wise difference in the embeddings.

moving edges between nodes randomly. Positive samples are generated with lower probability of changing, and nodes are more likely to be replaced by similar types, e.g. replacing FOR with While. Negative samples are generated with higher changing probability and randomness.

3.4 Mapping function

The graph embedding generated by the previous steps captures not only semantic information of the source code, but also the structure style of AST of the certain language. In the monolingual case, embeddings from different code fragments share the same language style information, thus the difference between embedding pairs can be regarded as semantic difference. But in cross-language situation, the language-wise difference is too significant to be ignored. However, according to our expectation, though code embeddings from different languages are distributed separately due to the structural difference of AST, they should obey same local distribution rule according to their semantics. Thus, we further proposes a mapping function that attempts to map the latent representation of one language to the other in order to blur the language-wise difference.

4 Experiments & Results

In this section we present the experiments we conducted and the current result of our models on both monolingual and cross-language code-clone detection. Though we did not achieve a promising result on the cross-language task, we tried to prove the feasibility of our approach, thus we argue that applying GNN models on such task is an idea worth trying, and there is much space for improvement for our model.

4.1 Monolingual clone detection

4.1.1 Experiment settings

We believe being able to detect monolingual code-clone is a precondition for a model to detect cross-language code-clones, so we started from the former task. We used two data sets for the monolingual code-clone detection task, one crawled from LeetCode[9] containing answers written in Python and Java for 100 different questions, and the data set from BigCloneBench[10] benchmark.

For the data set crawled from LeetCode, we injected positive and negative samples by data augmentation, with adding, removing and switching of nodes (sub-trees) by a specific probability. During the training and testing phase, only the original sample and injected positive and negative samples are labeled as positive and negative pairs respectively. We will discuss about this limitation in section 6.We used 70% of the data set for training, 15% for testing and 15% for validation.

BigCloneBench is a widely used large code clone benchmark that contains over 6,000,000 true clone pairs and 260,000 false clone pairs from 10 different functionalities extracted from Java projects. BigCloneBench mainly contains Type-3 and Type-4 clones, thus is appropriate for testing semantic code-clone detection models. In this work, instead of separating the data set of BigCloneBench into training and testing data set, we trained the model with the LeetCode data set and tested it with BigCloneBench, in order to prove the generalization ability of the proposed model.

We implemented the DeepWalk and GraphSage model following the source code from github repository[11,16]. We refer our models implementing DeepWalk and GraphSage as DeepWalk and GraphSage respectively, in the remainder of this paper. We used 3-layer MLPs with activation function ReLU for the readout function, gate function and aggregation function in the final aggregation step.

We train the model for 20 epochs with adaptive learning rate (the decrease speed of loss becomes slow enough according to observation). We compute the similarity score with normalized Euclidean distance and set the threshold to 0.6 for LeetCode data set and 0.8 for BigCloneBench (code pairs with score lower than 0.8 are considered clone).

4.1.2 Results

On the crawled LeetCode data set, our model managed to detect all injected positive clones, while classifying some of the negative pairs as clones. However, we have to mention that in the augmented data set, only the original injected samples will be selected as positive or negative pairs, which could cause bias on the result.

For test on the BigCloneBench, we used the testing tool BigCloneEval provided by the author. Though trained with a totally different data set, our model still managed to detect most of the Type-

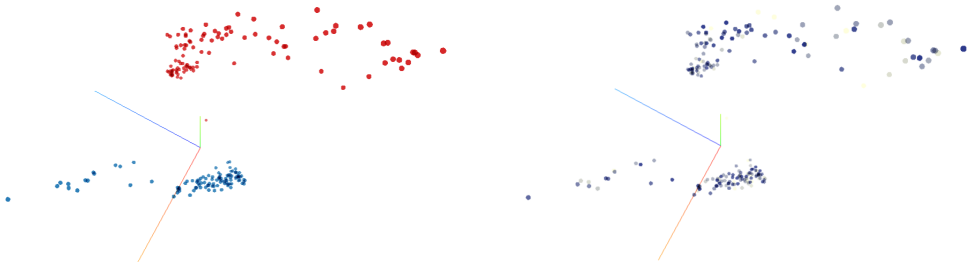
Table 1 Monolingual result on crawled LeetCode data set

model	precision	recall
DeepWalk	0.88	1
GraphSage	0.91	1

Table 2 Cross-language result on BigCloneBench

Recall Per Clone Type

Clone types	recall
Type-1	1.0
Type-2	1.0
Type-2 (blind)	1.0
Type-2 (consistent)	1.0
Very-Strongly Type-3	1.0
Strongly Type-3	0.9816
Moderately Type-3	0.8907
Weakly Type-3/Type-4	0.5117



(a) Language-wise distribution

(b) File name distribution

Figure 3 An example of the code embedding generated by the GraphSage model. (a) shows the distribution of code fragments from different languages, where red dots stand for Java and blue dots stand for Python. (b) shows the distribution of code fragments by their file index, where darker dots corresponds to greater index, however, file index has no direct relationship to the functionality of code fragment. While language-wise clustering is obvious, it's almost impossible to observe any distribution patter inside the clusters.

3 clones, failing mostly on weakly Type-3/Type-4 clones. This proves the feasibility of our model on monolingual code-clone detection task despite the bias on the training data set.

4. 2 Cross-language clone detection

4. 2. 1 Experiment settings

For detecting cross-language code-clones, first we project the code fragments from each programming language to independent latent spaces with the embedding generation model. Again we used DeepWalk and GraphSage respectively as the node embedding model. The experiment setting of this stage exactly follows the monolingual setting. Then we map one of the latent space into the other with a mapping function. We used a simple 3-layer MLP as the mapping function. In order to The embedding model and mapping function is trained separately. Finally, for a pair of randomly sampled code fragments, we compute the distance of their embedding and compare it with the threshold to judge if they are clone or not. For performance evaluation, we randomly sampled positive and negative code pairs at a ratio of 1:5, and computed Area Under the Curve (AUC) score.

4. 2. 2 Results

It's a pity to admit we did not acquire promising result in the experiment for detecting cross-language code-clones.

The training of code embedding model finished normally. We generated code embedding with model implementing DeepWalk and GraphSage respectively. However, the training of mapping function on both embedding set failed to converge. The loss of mapping function on both embedding set oscillate around 0.4-1.5. In the evaluation, while the model implementing DeepWalk got an AUC score of 0.5, the AUC score for the model implementing GraphSage occasionally reaches 0.58-0.6, indicating it works slightly better by capturing individual structure of code fragment.

For the embedding generated by the GraphSage model, we put the vector representation of Python and Java code together to observe the pattern. The model generated clearly separated embedding for Python and Java, which means the model is capable of distinguishing language-wise structure difference even without explicit training. But there is no

sign of specific pattern inside each language cluster which is required for code-clone detection.

5 Related work

The literature [7] is the most recent work trying to solve monolingual code-clone detection task with GNN model. As the first try on detecting code-clones with GNN, they reached very promising result with their proposed model. Instead of working directly on ASTs, they designed a novel graph representation form FA-AST for Java programs by adding control flow edges and data flow edges to enrich potential information contain in the original ASTs. For the GNN model, they applied GGNN [3] and graph matching network GMN [2] as comparison model, and finally reached better result with the GMN model.

Unlike typical GNN models that generate node embedding or graph embedding for a single graph each inference time, GMN model takes a pair of graphs as input each time, and conducts a pair-wise classification. Due to the pair-wise attention mechanism, a GMN model is capable of capturing general similarity of a pair of graphs while focusing on detailed key differences. Thus by applying the GMN model, the literature [7] managed to detect structurally similar negative samples and positive samples that only share detailed similarity.

However, as a monolingual code-clone detector, [10] requires well labeled data set to work properly. Besides, since GMN is a pair-wise classifier, the embedding it generated for each graph depends on the input pair thus each inference process requires a complete work procedure of the whole model, and the generated embeddings cannot be reused in other situation. For the cross-language code-clone task, the graph structure for ASTs from different languages differs too much for the pair-wise attention mechanism to work correctly.

6 Conclusion & Future work

As proved in the literature [7], GNN models can be a powerful tool for monolingual code-clone detection. In our work, we would like to further argue that GNN is also a proper choice for cross-language code-clone detection. By the preliminary experiments and results, we show that a GNN model can project the code fragments into a latent space, but the information preserved by that projection greatly depends on the data set and the design of training stage.

Currently in our proposed model:

- the possible range of input sampling is severely limited by the augmented data set
- what the model can learn from the data set is greatly biased due to the sub-optimal rule of data augmentation
- the mapping function is too simple that might not be capable of blurring the language-wise difference

Despite these unsolved problems, we still insist that a proper embedding model for generating language-irrelevant embeddings can be proposed base on GNN models. In the future, we will focus on overcoming the listed difficulties to improve the performance of our model.

参 考 文 献

[1] Hamilton, W., Ying, Z., and Leskovec, J.: Inductive representation learning on large graphs, *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[2] Li, Y., Gu, C., Dullien, T., Vinyals, O., and Kohli, P.: Graph matching networks for learning the similarity of graph structured objects, *arXiv preprint arXiv:1904.12787*, (2019).

[3] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R.: Gated graph sequence neural networks, *arXiv preprint arXiv:1511.05493*, (2015).

[4] Nichols, L., Emre, M., and Hardekopf, B.: Structural and Nominal Cross-Language Clone Detection, *Fundamental Approaches to Software Engineering*, Hähnle, R. and van der Aalst, W.(eds.), Cham, Springer International Publishing, 2019,

pp. 247–263.

[5] Perez, D. and Chiba, S.: Cross-language clone detection by learning over abstract syntax trees, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 518–528.

[6] Perozzi, B., Al-Rfou, R., and Skiena, S.: Deepwalk: Online learning of social representations, *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[7] Wang, W., Li, G., Ma, B., Xia, X., and Jin, Z.: Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree, *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 261–271.