Source Code is a Graph, Not a Sequence: A Cross-Lingual Perspective on Code Clone Detection

Mohammed Ataaur Rahaman[1], **Julia Ive**[1]
Queen Mary University of London[1]
m.a.rahaman@se22.qmul.ac.uk
j.ive@qmul.ac.uk

Abstract

Code clone detection is challenging, as source code can be written in different languages, domains, and styles. In this paper, we argue that source code is inherently a graph, not a sequence, and that graph-based methods are more suitable for code clone detection than sequence-based methods. We compare the performance of two state-of-the-art models: CodeBERT (Feng et al., 2020), a sequence-based model, and CodeGraph (Yu et al., 2023), a graph-based model, on two benchmark data-sets: BCB (Svajlenko et al., 2014) and PoolC (PoolC, no date). We show that CodeGraph outperforms CodeBERT on both data-sets, especially on cross-lingual code clones. To the best of our knowledge, this is the first work to demonstrate that using graphs is more effective than sequences for identifying similar code written in different languages.

1 Introduction

Existing methods for code clone detection can be broadly classified into two categories: sequence-based and graph-based. Sequence-based methods rely on textual similarity of the code, such as token sequences. Graph-based methods rely on structural similarity of the code, such as Abstract Syntax Tree (ASTs), or control flow graphs (CFGs) or Code Property Graphs (CPGs). Sequence-based methods are fast and scalable, but they may fail to detect clones that have different syntax or structure. Graph-based methods are more accurate and robust, but they may be slow and complex, especially for large-scale or cross-language code clone detection.

A python source code clone pair is presented in Listing [1, 2]. The two code snippets have the same semantic behavior: they print 'A' or 'a' depending on the case of the input. However, they differ in their syntactic forms. Further such examples can be viewed in Appendix C.

In this paper, we argue that source code is naturally a graph, not a sequence, and that graph-based methods are more suitable for code clone detection than sequence-based methods. We compare the performance of sequence-based and graph-based methods for code clone detection on two benchmark data-sets: BCB (Svajlenko et al., 2014) and PoolC (PoolC, no date). BCB is a data-set of Java code snippets where as PoolC is a data-set of Python code snippets. We use CodeBERT (Feng et al., 2020) as a representative sequence-based modelling approach, and CodeGraph (Yu et al., 2023) as a representative graph-based modeling approach. CodeBERT is a bimodal pre-trained model for programming language (PL) and natural language (NL) that learns general-purpose representations that support downstream NL-PL applications. CodeGraph is a graph-based model for semantic code clone detection based on a Siamese graph-matching network that uses attention mechanisms to learn code semantics from DFGs and CPGs.

```
S = input()

if S.isupper():
  print("A")
else:
  print("a")
```

Listing 1: Python code 1

```
alp=input()

if alp==alp.upper():
  print("A")
elif alp==alp.lower():
  print("a")
```

Listing 2: Python code 2

We conduct various experiments to evaluate the accuracy, recall, precision, and F1-score of CodeBERT and CodeGraph on three experimental setups: (i) in-domain static source code analysis , (ii) cross-lingual generalization and semantic extraction, and (iii) zero-shot source code clone classification. We show that CodeGraph outperforms CodeBERT across experimental setups and metrics. The main contributions of this paper are as follows:

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop), pages 168–199

- To best of our knowledge, we are the first one to demonstrate the superiority of graph-based methods over sequence-based methods for multilingual static source code analysis tasks, such as clone detection, by exploiting the natural graph structure of source code across programming languages.

- We provide novel insights on the generalization and cross-domain understanding of graph-based models, compared to sequence-based models, for source code analysis, as they leverage both the syntactic and semantic features of source code in various cross-domain settings.

- We show how mixing cross-lingual data-sets can improve the overall performance of the graph-based model by 4.5%, as it can learn from the commonalities and differences between programming languages.

- We focus on the so far under-explored clone detection Python data-set PoolC, along with the benchmark Java data-set BCB, and draw parallel comparisons on both of the data-sets.

2 Related Work

2.1 Sequence based modeling

There has been various sequence based modelling approaches used by source code clone detection like, CodeBERT (Feng et al., 2020), UNIXCODER (Guo et al., 2022), ContraBERT (Liu et al., 2023). Here in sequence modeling the source code is tokenized as a piece of words (or source code). This tokenized pieces of words in a sequence is learnt by the model to understand a fragment of code. This helps the model learn the semantics, by taking the code in a sequential manner.

We use CodeBERT (Feng et al., 2020) which is a bimodal pre-trained model for programming language (PL) and natural language (NL) that learns general-purpose representations that support downstream NL-PL applications such as natural language code search, code documentation generation, etc1. CodeBERT is developed with a Transformer-based neural architecture, and is trained with a hybrid objective function that incorporates the pre-training task of replaced token detection, which is to detect plausible alternatives sampled from generators 1, along side with Masked Language modelling. In this study, we use CodeBERT as a pre-trained model for our sequence model for source code clone detection.

2.2 Graph based modeling

On the other side, clone detection as a graph modelling approach, we have models like TBCCD (Yu et al., 2019), FA-AST (Wang et al., 2020), HOLMES (Mehrotra et al., 2020), DG-IVHFS (Yang et al., 2023), CodeGraph4CCDetector (Yu et al., 2023). These types of graph models first construct a tree or a graph like, abstract syntax tree, Control flow graph etc from the source code. This helps to retain the structural information of the code, regardless of it being moved from its location or variables being replaced. This ideally should help the model concentrate more on the semantics, rather than the structural learning, as it is already baked into its structure.

We use CodeGraph4CCDetector (Yu et al., 2023) as our graph-based model, from here on referred as CodeGraph. This model is reported to have state of the art results on the BCB (Svajlenko et al., 2014) data-set. This is a Siamese graph matching network which basically takes in two source code snippets and output a similarity score between them. The input for this is the Code Property Graph, which is essentially graph having various nodes and edges. This helps the network capture the source codes syntactical and semantical information. The node representation of this CodeGraph uses attention mechanism on a node level to extract out a node representation, before combining it to graph level representation. The major advantage of a graph level over the sequence level is, this can handle code snippets of different lengths and structures, as long as the hardware memory can load it.

3 Methods

In this section, for the source code representations, two methods are employed: byte pair tokenization (Sennrich et al., 2016) and code property graphs (CPGs). Byte pair tokenization is used to sequence source code into tokens using CodeBERT's BPE tokenizer, while CPGs represent source code as graphs, combining abstract syntax trees (ASTs) and data flow graphs (DFGs) into a unified graph. Tree-sitter, a lexical parser, generates ASTs for various languages, and Microsoft's DFG generator (Guo et al., 2020) adds data flow edges to these ASTs. The CPGs are then standardized across languages by pruning non-essential nodes and standardizing

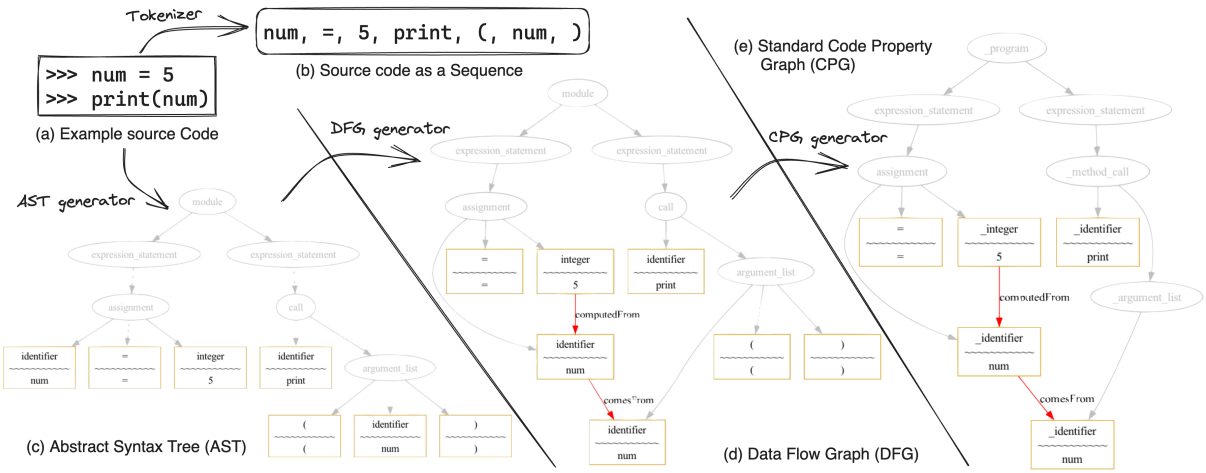


Figure 1: This figure illustrates how the source code can be transformed into a sequence and a graph. (a) A sample Python program that prints a number, 5. (b) The code tokens using CodeBERT's BPE tokenizer. (c), (d), and (e) are the graph representations of the code as Abstract Syntax Tree, Data Flow Graph, and Standard Code Property Graph, respectively. This shows how Standard CPG (e) is the most concise and standardized graph representation across languages, compared to raw, AST, or DFG.

node type labels. This standardization allows for consistent recognition of code structures across languages, proving advantageous for code clone detection.

We use two models for source code clone detection: CodeBERT, a sequence-based model, and CodeGraph, a graph-based model. CodeBERT is fine-tuned on a binary classification task to determine if a pair of source codes are clones, capturing syntactic and semantic information through pre-training on multiple languages. More details in Appendix B.3. CodeGraph employs a trained word2vec model (Mikolov et al., 2013) to generate token embeddings, maintaining consistency with CodeBERT. Both models process code pairs to produce representations used for binary classification, with CodeGraph utilizing an LSTM layer for graph-level representation analysis. More details in Appendix B.4.

4 Experimental Design

Based on our proposed methodology, we conduct experimentation on the following research questions (RQs):

- RQ1: Will a graph-based model that leverages both structural and semantic information surpass a sequence-based model in an in-domain static source code analysis?
- RQ2: Will a graph-based model trained on multiple source code languages outperform a sequence-based model in cross-lingual generalization and semantic extraction?
- RQ3: Will a graph-based model excel over a sequence-based model in the domain generalization of zero-shot source code clone classification?

Please note, within our scope, "multi-lingual" pertains to experiments conducted across a range of programming languages. "Cross-lingual", on the other hand, denotes the concurrent utilization of two programming languages, where the dataset comprises a blend of both languages. This allows the model to process and interpret the mixed language data in tandem.

4.1 Experiment Data

For our experimental setups, we perform clone detection on two publicly available data-sets. The first one is Big Clone Bench (BCB), which is a java language data-set that was originally introduced by Svajlenko et al. (2014). We used the version of BCB that was filtered according to FA-AST (Wang et al., 2020). BCB contains 9,134 Java methods, which generate over 2M combinations of clone and non-clone code pairs. The second one is PoolC, which consists of over 6M python code snippets that were extracted from hugging face (PoolC, no date). Our manual inspection has confirmed the reliability and usefulness of this data-set for our experimental purposes. This data-set has so far been

Attribute	BCB (Java)	PoolC (Python)	Mix_1 (Java + Python)
Actual File Counts	9,126	44,950	-
Filtered File Counts	2,048	17,570	19,063
Avg* Lines	12	10	10
Avg* Characters	450	158	190
Avg* Tokens	200	83	96
Avg* Nodes	76	67	68
Avg* Leaf Nodes	36	32	32
Avg* AST Edges	75	66	67
Avg* DFG Edges	15	22	21

*Avg: Average on the filtered files.

Table 1: Data-set counts of actual and filtered file counts, with their static metrics.

under-exploited.

For environmental reasons, during the experimentation phase we randomly sampled pairs of clone and non-clone from the filtered files set to form the data-set (see Appendix B.5 and G.1 for more details). Table 5 summarizes the data-set pairs according to each data-set.

4.2 Experimental Setup

We chose the state-of-the-art sequence model and graph model, namely CodeBERT (Feng et al., 2020) and CodeGraph (Yu et al., 2023), respectively, to conduct various experiments. To answer the research questions, we designed the experiments around them as follows.

- Experiment 1: We train and evaluate Sequence and Graph models independently on each of the data-sets, namely BCB and PoolC, to compare their performance within the same domain as baselines.
- Experiment 2: We train and evaluate Sequence and Graph models on Mix_1 Data-set, which is a mixture of data from both domains, to examine their cross-domain learning and generalization capabilities.
- Experiment 3: We train Sequence and Graph models on BCB data-set and test them on PoolC data-set, and vice versa, to assess their cross-domain zero-shot performance.

4.3 Model Hyper-parameters

We use the same machines with Intel® Xeon® Gold 5222 and one Quadro RTX 6000 to train both the sequence and graph models (CodeBERT and CodeGraph, respectively) in order to maintain a consistent experimentation environment. The maximum batch size that CodeBERT can run on a single RTX 6000 is 16 code pairs, or 32 code snippets per batch. We also set the batch size of CodeGraph to the same value. The other hyper-parameters used for training these models are given in Appendix E.

5 Results

5.1 Experiment 1

We train the sequence and graph models (CodeBERT and CodeGraph, respectively) on two datasets: BCB and PoolC. This leads to four model trainings and evaluations, as shown in Table 2. We select the best-performing epochs for each model, which are the 3rd epoch for CodeBERT and the 2nd epoch for CodeGraph. We find that CodeGraph consistently outperforms CodeBERT on both datasets, demonstrating that CodeGraph has a better learning capability on the source code than CodeBERT under limited data and constrained environment conditions. We highlight statistically significant experimental results in the tables based on bootstrap testing (Fornaciari et al., 2022) with p value below 0.05 for statistical significance, which compares CodeBERT and CodeGraph.

Answer to RQ1: Baseline on the BCB and PoolC data-sets, suggests that the graph based model outperforms the sequence based model. This suggests that the graph model can better capture the structural and semantic information of the source code than the sequence model.

Model Name	Train & Eval Dataset	Metrics			
		A	*P*	*R*	*F1*
CodeBERT	BCB	97.62	97.63	97.62	97.62
CodeGraph	BCB	**98.88***	**98.88***	**98.88***	**98.87***
CodeBERT	PoolC	81.82	83.92	81.82	81.54
CodeGraph	PoolC	**84.00***	**84.86**	**84.00***	**83.90***

A: accuracy, P: precision, R: recall, F1: F-score

Bold is best value, * is statistically significant.

Table 2: Experiment 1 | Performance of Graph-Based and Sequence-Based Models on BCB and PoolC Data-Sets.

5.2 Experiment 2

We use the same model architectures from Experiment 1, but we train them on a cross-lingual data-

set (Mix_1) that combines both the BCB and PoolC data sets. We then evaluate these models on the Mix_1 data-set as well as the individual BCB and PoolC data-sets. The results are shown in Table 3. The evaluation results on the Mix_1 dataset for both CodeBERT and CodeGraph are intermediate between the single-language models trained in Experiment 1. This is further confirmed by the evaluation results on the individual BCB and PoolC data-sets, where we observe that cross-lingual training improves the performance of CodeGraph on both data-sets, from 83.90 to 87.64 F1 on the PoolC data-set and from 98.87 to 99.42 F1 on the BCB data-set, indicating that CodeGraph generalizes better on the source code with cross-lingual training. On the other hand, we observe that cross-lingual training does not improve the performance of CodeBERT as much as CodeGraph, decreasing it by -0.55 F1 on the BCB data-set and increasing it by only +0.19 F1 on the PoolC data-set.

Answer to RQ2: The results on the cross-lingual setting of CodeBERT and CodeGraph models, i.e. trained on Mix_1 data-set, demonstrate that CodeGraph is a more generalized model than CodeBERT as evidenced by the improvement in the performance of CodeGraph especially on PoolC data-set, whereas we observe a decline in the performance of CodeBERT on BCB data-set and marginal improvement on PoolC dataset. This implies that graph models are more adaptable for cross-lingual source code analysis.

Model Name	Train Dataset	Eval Dataset	Metrics			
			A	*P*	*R*	*F1*
CodeBERT	Mix_1	Mix_1	90.35	90.51	90.35	90.34
CodeGraph			**93.65***	**93.77***	**93.65***	**93.65***
CodeBERT	Mix_1	BCB	97.08	97.11	97.08	97.07
CodeGraph			**99.42***	**99.43***	**99.42***	**99.42***
CodeBERT	Mix_1	PoolC	81.82	82.53	81.82	81.73
CodeGraph			**87.68***	**88.13***	**87.68***	**87.64***

A: accuracy, P: precision, R: recall, F1: F-score

Bold is best value, * is statistically significant

Underlined is statistically significant & better w.r.t Experiment 1.

Table 3: Experiment 2 | Performance of Graph-Based and Sequence-Based Models on Mix_1 Data-Set.

5.3 Experiment 3

We test the domain generalization of the pre-trained models from Experiment 1, i.e., CodeBERT and CodeGraph, on a different source code language than the one they were trained on. For instance, we evaluate CodeBERT trained on BCB on PoolC, and vice versa. We repeat the same procedure with CodeGraph without changing the experimental setup. The results of this experiment are shown in Table 4.

This experiment simulates the domain generalization from Python source code to Java source code and vice versa. The results show that CodeBERT performs very poorly on a different domain, with F1 scores of 33.71 and 36.56 for PoolC and BCB evaluation, respectively. This indicates that the model has over-fitted on the domain and cannot generalize well to a new domain. We observe the same trend with more epochs. On the other hand, CodeGraph performs much better than CodeBERT on a different domain, with F1 scores of 53.67 and 46.44 for PoolC and BCB evaluation, respectively. This demonstrates that CodeGraph has a better domain generalization capability than CodeBERT in a zero-shot learning setting, although it does not achieve state-of-the-art performance. This suggests that representing source code as a graph rather than a sequence is a promising direction for future research.

Answer to RQ3: The results on the domain generalization task show that CodeGraph outperforms CodeBERT in adapting to a new source code language domain without any labeled data for that domain during training. This indicates that graph-based model has an advantage over sequence-based model in the domain generalization of zero-shot source code clone classification task.

Model Name	Train Dataset	Eval Dataset	Metrics			
			A	*P*	*R*	*F1*
CodeBERT	BCB	PoolC	50.05	53.58	50.05	33.71
CodeGraph			**53.67**	**53.68**	**53.68***	**53.67***
CodeBERT	PoolC	BCB	48.95	45.20	48.95	36.56
CodeGraph			**54.88***	**63.16***	**54.88***	**46.44***

A: accuracy, P: precision, R: recall, F1: F-score

Bold is best value, * is statistically significant.

Table 4: Experiment 3 | Performance of Graph-Based and Sequence-Based Models on Cross-Domain Zero-Shot Evaluation.

5.4 Discussion

We analyze the false predictions made by both the models, CodeBERT and CodeGraph, and find that most of them are false positives, especially

from CodeBERT. Furthermore, when we examine these examples from CodeBERT, we notice that the model predicts them as false positives with high confidence, whereas the CodeGraph model either predicts them as true negatives or false positives with low confidence. This indicates that adjusting the classification threshold for CodeGraph could even further improve its overall performance. However, for CodeBERT, we observe that the model exhibits difficulty distinguishing between code snippets when identical tokens or keywords are present, even if they serve different semantic purposes. This often results in the model erroneously identifying non-clone pairs as clones due to superficial lexical similarities.. We provide a detailed analysis of this in Appendix F.2.

We also analyze the false negatives for CodeGraph on the PoolC data-set, which are the most frequent among all the models and data-sets. We find that these false negatives are mainly due to the large size differences between the code pairs in the PoolC data-set. The examples we inspect are clones of type IV, but they have one code snippet much longer than the other. This makes it difficult for CodeGraph to recognize them as clones and it predicts them as non-clones instead. We provide some detailed explanation and examples of these false negatives in Appendix F.3.

A significant factor that appears to contribute to the superiority of the graph-based approach over the sequence-based method is the visual similarity of Code-Property Graphs across various programming languages, as illustrated in Figure 2 and elaborated upon in Appendix B.2.3.

6 Future Research

Some possible directions for the future research based on the limitations G.1 are as follows:

- To evaluate the impact of data-set size on the performance of the models, future research could use the complete and more diverse data-set that include source code files with more than 100 nodes and data-set samples itself going upwards of a million samples. This would help to test the generalizability and robustness of the models across different domains and languages at a larger scale.
- Train a mixture model on various source code languages, not just limiting to two, such as JavaScript, SQL, HTML, etc., and evaluate its generalization ability on different domains together. Moreover, cross-domain example pairs could be generated from Code Forces (Yeo, 2023), which is an online platform for competitive programming that supports multiple languages.

7 Conclusion

In this paper, we have shown that graph-based methods are superior to sequence-based methods for source code clone detection. We have used the state-of-the-art models CodeBERT (Feng et al., 2020) and CodeGraph (Yu et al., 2023) to conduct various experiments on two benchmark data-sets: BCB (Svajlenko et al., 2014) and PoolC (PoolC, no date). We have demonstrated that graph models can better capture the structural and semantic information of the source code than sequence models in a series of 3 experimental setups, and that they can generalize better across different source code languages and domains. We have also provided efficient and scalable code for generating standard CPG representations of source code, along with the re-implemented code for the sequence and graph-based models. Our work has important implications for future research on source code analysis, as it suggests that representing source code as a graph rather than a sequence is a promising direction for enhancing the performance and generalization of static source code analysis models.

References

Barry W. Boehm. 1981. Software engineering economics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Tommaso Fornaciari, Alexandra Uma, Massimo Poesio, and Dirk Hovy. 2022. Hard and soft evaluation of NLP models with BOOtSTrap SAmpling - BooStSa. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 127–134, Dublin, Ireland. Association for Computational Linguistics.

Google. no date. Google code jam dataset.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun

Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366.

Melina Kulenovic and Dzenana Donko. 2014. A survey of static code analysis methods for security vulnerabilities detection. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1381–1386.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pretrained models via contrastive learning.

Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2020. Modeling functional similarity in source code with graph-based siamese networks. *CoRR*, abs/2011.11228.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space.

PoolC. no date. Poolc/1-fold-clone-detection-600k-5fold.

Radim Rehurek and Petr Sojka. 2011. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2).

Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480.

Tree-Sitter. no date. Parser generator tool.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. *CoRR*, abs/2002.08653.

Haixin Yang, Zhen Li, and Xinyu Guo. 2023. A novel source code clone detection method based on dual-gcn and ivhfs. *Electronics*, 12:1315.

Geremie Yun Siang Yeo. 2023. Dataset and Code for: Code problem similarity detection using code clones and pretrained models.

Dongjin Yu, Quanxin Yang, Xin Chen, Jie Chen, and Yihang Xu. 2023. Graph-based code semantics learning for efficient semantic code clone detection. *Inf. Softw. Technol.*, 156(C).

Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80.

A Related Work

A.1 What is static source code analysis?

Static code analysis is a valuable technique for improving software quality and security without actually compiling the code. It can find errors that are hard to detect at run time, improve the quality and maintainability of the code, and reduce the cost and time of testing and debugging. This is basically a form of white-box testing. According to Boehm (1981) the cost of fixing a defect increases exponentially as it moves from the coding phase to the testing phase to the maintenance phase. Therefore, having a static tools to analyse and fix a source code as soon as possible helps save lot of resources and efforts.

A.2 What are applications of static code analysis?

There are various different applications for static code analysis. Security Vulnerability detection, is one of the major static code analysis, which can help developers identify and fix security vulnerabilities before they are exploited by the attackers as extensively stated by Kulenovic and Donko (2014). Another important area of static code analysis is in helping developers find inefficient algorithms and improve the resource utilization, response time, and other throughput of the software. Clone Detection is another application that can help identify similar functional code fragments which may indicate code duplication, plagiarism or reuse. This can improve quality, maintainability and the security of the code by eliminating redundant and inconsistent or outdated code (Roy and Cordy, 2007).

A.3 What is source code clone detection?

Source code clone detection is the process of finding code fragments that have similar functionalities or structures, which could indicate that the code is a duplicate, plagiarised or reused, which may be done purposefully, negligently or accidentally by a developer as said by Roy and Cordy (2007). Clone detection is very harmful to the quality of the entire source code (Roy and Cordy, 2007). A broad categorization on various types of code clones is given by (Roy and Cordy, 2007).

- **Textual Similarity**
  - **Type I**: Changes in White-spaces, comments, layouts.
  - **Type II**: Renaming of variable names, or changes in types and identifiers.
  - **Type III**: Addition or removal of statements.
- **Functional Similarity**
  - **Type IV**: Complete change in syntax, but functionally same behavior.

A.4 Types of Clone detection approaches?

According to Roy and Cordy (2007) there are multiple techniques to detect a source code clones, like Text based, token based, tree based, Program dependency graph based, metrics based, or hybrid approaches. Here we deal and compare token based vs tree based clone detection approach. As we know that, to detection semantically same code clones, the model should not just rely on difference of syntax, but also understand the semantics of the structure. This becomes hard, if we pass the source code to the model as sequence rather than a Tree like Abstract Syntax Tree which naturally holds the syntatical information of a source code.

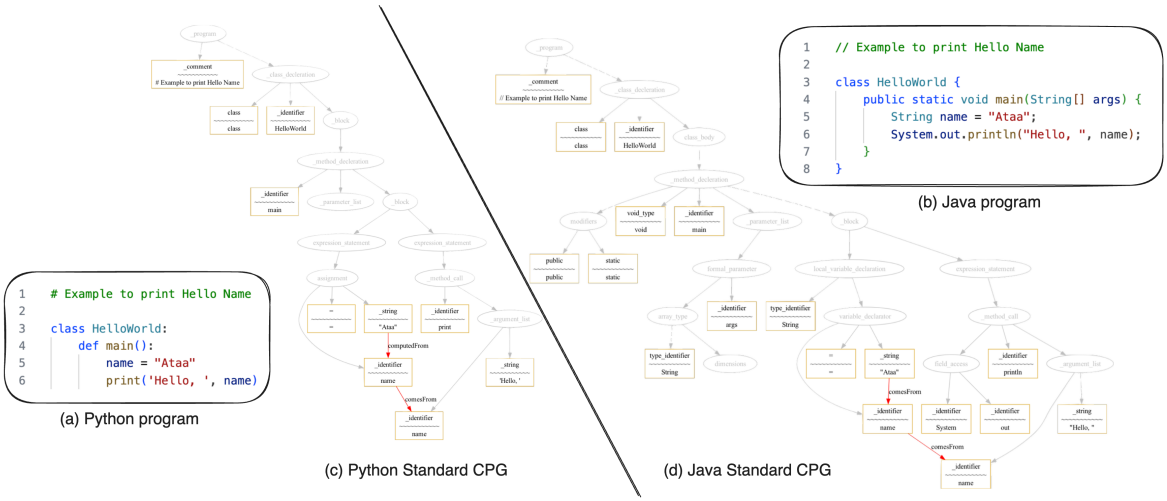


Figure 2: (a) and (b) show the Java and Python programs that print hello, name, respectively. (c) and (d) show the corresponding standard CPGs, which are generated by applying lexical parsing, data flow generation, and graph standardization to the source code. The standard CPGs look identical for both languages, as they capture the common structure and logic of the programs.

B Methods

This section describes the methods and models that we employ for our experiments on the sequence and graph representations of source code. We organize this section into four parts. First, we present how we use byte pair tokenizers to represent source code as a sequence of tokens. Second, we illustrate how we use code property graphs (CPGs) to represent source code as a graph. Third, we introduce CodeBERT (Feng et al., 2020), the sequence-based model that we employ for code clone detection. Fourth, we present CodeGraph (Yu et al., 2023), the graph-based model that we employ for code clone detection.

B.1 Code Tokenization

We apply the default Byte Pair Encoding, BPE tokenizer of CodeBERT (Feng et al., 2020) to represent the source code as a sequence of tokens. Figure 1b shows an example of how the BPE tokenizer splits the source code in Figure 1a into word and subword tokens.

B.2 Code Representations

We use a graph representation of source code that consists of nodes and edges connecting source code tokens. To generate this representation, we apply the following steps: First, we use a lexical parser, to produce the abstract syntax tree (AST) of the source code. Second, we extract the data flow information from the AST. Third, we merge the AST and DFG, to form one graph which we call it as Code Property Graph. This is further pruned and standardized across source code languages to make the standard CPG.

B.2.1 Abstract Syntax Tree (AST)

We use Tree-sitter (Tree-Sitter, no date), a lexical parser, to generate the abstract syntax tree (AST) of the source code for any language. We currently use it for Java and Python, but it supports 141 different languages. Figure 1c shows the AST generated from the sample source code in Figure 1a.

B.2.2 Data Flow Graph (DFG)

We use Microsoft's Data Flow Generator (DFG) (Guo et al., 2020) to generate a data flow graph (DFG). This DFG generator takes the AST from the previous step and adds the data flow edges to it to form the DFG. Figure 1d shows the DFG graph for the same example source code in Figure 1a. We can see that there is a data flow edge between the integer literal '5' and the variable 'num', as '5' is assigned to 'num'. There is also a data flow edge between the second occurrence of 'num' and the print statement, as 'num' is used as an argument.

B.2.3 Standard Code Property Graph (CPG)

We standardize the DFG to a code property graph (CPG), which is our final graph representation of source code. We perform two main operations to standardize the DFG across languages. First, we prune the graph from nodes that do not add value to the model's understanding, such as opening and closing brackets that are implicitly understood when there is a method call. Second, we standardize the node type labels across languages so that the model can recognize them consistently across languages. For example, in Figure 1e we see that the root node 'module' and 'integer' node are standardized and replaced with '_program' and '_integer' as standard node types.
The major impact of a standard CPG can be seen in Figure 2, where two programs that print hello, name in Java and Python are written. The programs look different as raw code, but they have the same functionality and semantics. The standard CPGs look very similar in both cases, as shown in Figure 2c and 2d. We provide more examples of standard CPGs in Appendix C. This supports our claim that this type of code representation is more suitable than the sequence of code for identifying code clones.

B.3 Sequence-based Model: CodeBERT

In order to model a sequence model for source code clone detection, we use CodeBERT (Feng et al., 2020) as a pre-trained model. We fine-tune CodeBERT on the source code clone detection labelled data-set. The fine tuning task is a binary classification task where the source code pair is passed sequentially through the CodeBERT which acts as an encoder, and the 2 representation vectors coming out from this encoder, is concatenated and passed to a shallow 2 layer MLP classifier to give the final output, if the pair is a clone or a no clone. The major advantage of using this state of the art encoder CodeBERT is that it can help capture both the syntactic and semantic information of the PL code, by leveraging the large-scale pre-training data of multiple languages.

B.4 Graph-based Model: CodeGraph

We use CodeGraph4CCDetector (Yu et al., 2023) as the graph based model for our source code clone detection, it is from here on refered to as CodeGraph model. This model initially is used by its authors on BCB data-set for its classification, and hence we keep the pipeline as it is. However, we trained our own word2vec model (Mikolov et al., 2013), using gensim (Rehurek and Sojka, 2011) to keep it consistent with respect to the sequence model. We call this model as Code2Vec model, which helps to generate the source code token embedding for our source code. We train this Code2Vec model using the source tokens which are tokenized by the CodeBERT's (Feng et al., 2020) tokenizer, which is a Byte Pair encoding tokenizer. This helps in two ways, Firstly, it helps to keep it consistent with the comparison of the sequence model, and secondly it helps to retain the word meanings of the human written source code variable names etc. Once we get the tokenized vector format of each graph nodes using the Code2Vec model on every node of CPG, we then use the CodeGraph architecture as it is. Here similar to the sequence model, we pass the code pair sequentially to the CodeGraph model, which then generates the graph level representation. This representation is the taken to a shallow LSTM layer (which is trained along with the graph model) helps to perform a binary classification on this graph level representation.

B.5 Dataset

We applied various parameters to limit the data-set for the experimentation phase. We restricted the number of lines to be between 5 and 100, the maximum number of characters to be 2000, and the maximum number of nodes in the graph to be 100. The details of how the distribution changed before and after applying the thresholds are given in Appendix D. Table 1 shows the summary of the average counts for the filtered files according to each data-set (BCB, PoolC, and their combination, Mix_1).

Dataset	Split	Total pairs	Positive	Negative
BCB	Train	50,855	29,070	21,785
	Test	4,000	2,000	2,000
	Val	4,000	2,000	2,000
PoolC	Train	50,500	25,250	25,250
	Test	4,000	2,000	2,000
	Val	4,000	2,000	2,000
Mix_1	Train	50,000	25,000	25,000
	Test	4,000	2,000	2,000
	Val	4,000	2,000	2,000

Positive: Clone pairs | Negative: Not a Clone pair.

Table 5: Data-set sample size. Equally sampled from each of the data-sets.

C Code Representations

C.1 Python Standard Code Property Graphs Example pairs

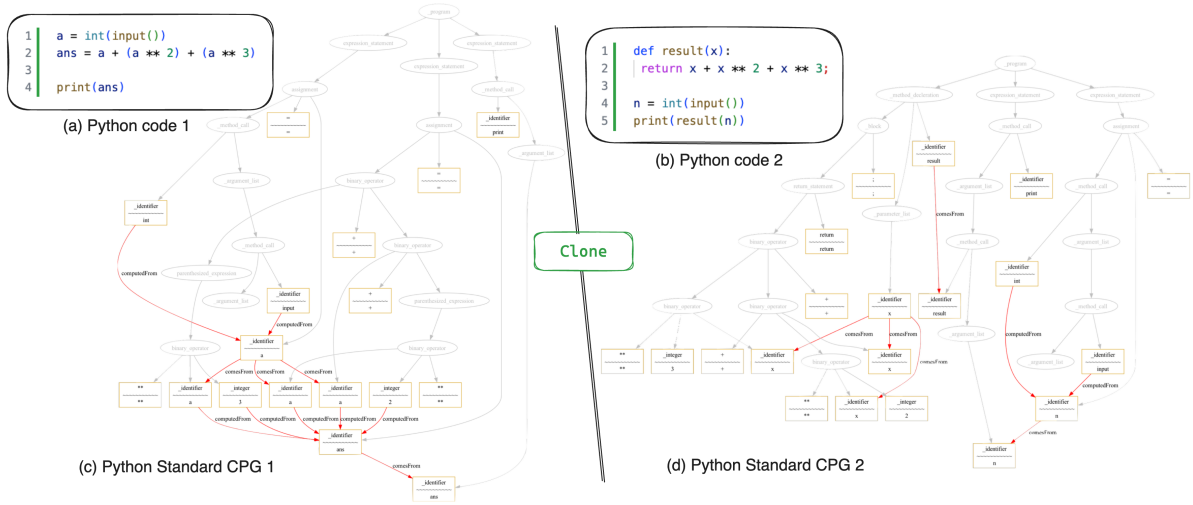


Figure 3: An example of python code clone pairs with its Standard Code Property Graphs. (a) & (b) are the source codes, and (c) & (d) are its respective Standard Code Property Graphs.

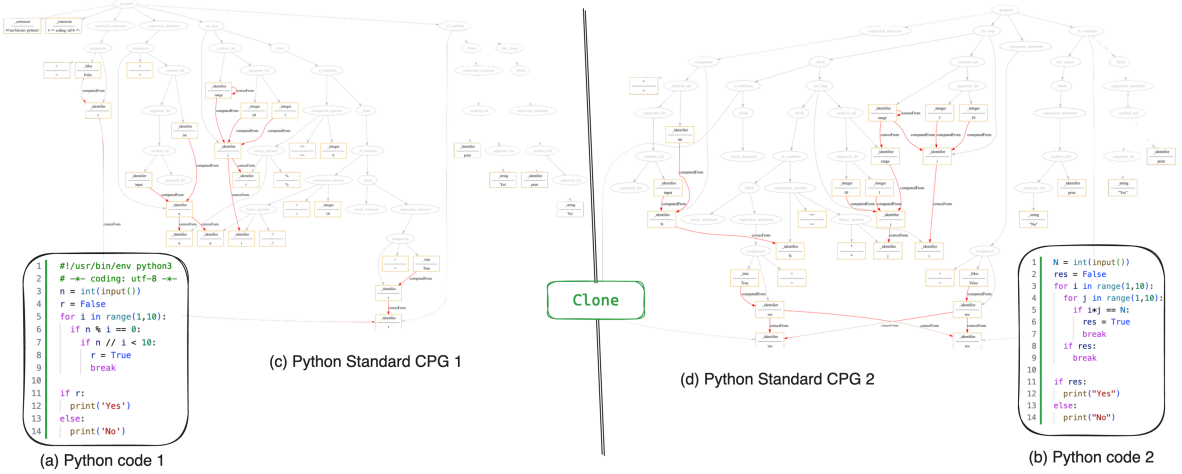


Figure 4: An example of python code clone pairs with its Standard Code Property Graphs. (a) & (b) are the source codes, and (c) & (d) are its respective Standard Code Property Graphs.

C.2 Java Standard Code Property Graphs Example pairs

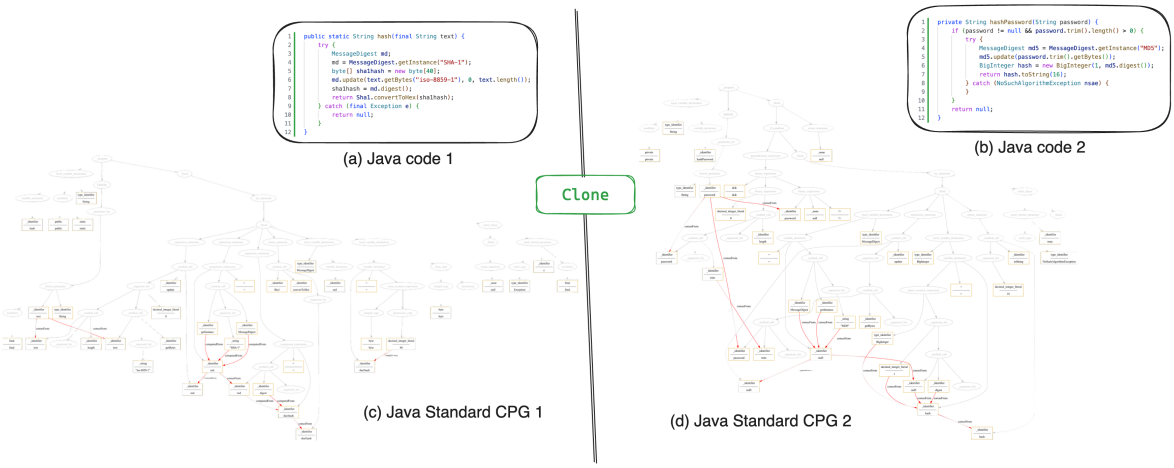


Figure 5: An example of Java code clone pairs with its Standard Code Property Graphs. (a) & (b) are the source codes, and (c) & (d) are its respective Standard Code Property Graphs.

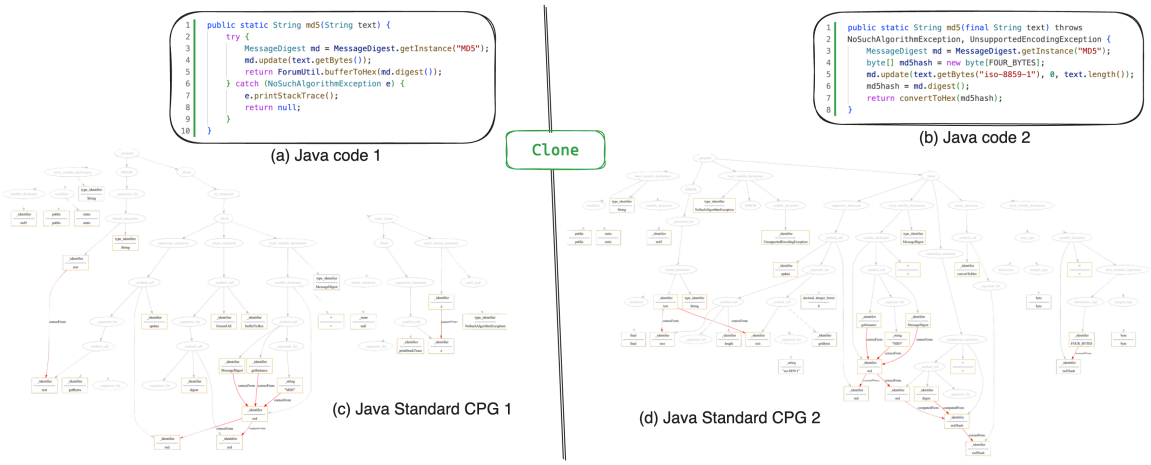


Figure 6: An example of Java code clone pairs with its Standard Code Property Graphs. (a) & (b) are the source codes, and (c) & (d) are its respective Standard Code Property Graphs.

D Data-set

Data set is filtered based on various parameters like number of lines, number of characters, and number of nodes. Given below are the charts how the data looks before and after filtering.

D.1 Java Data : BCB

Given in Figure 10 are the original and filtered distributions for Java dataset from BigCloneBench BCB dataset (Wang et al., 2020).

[b]0.32

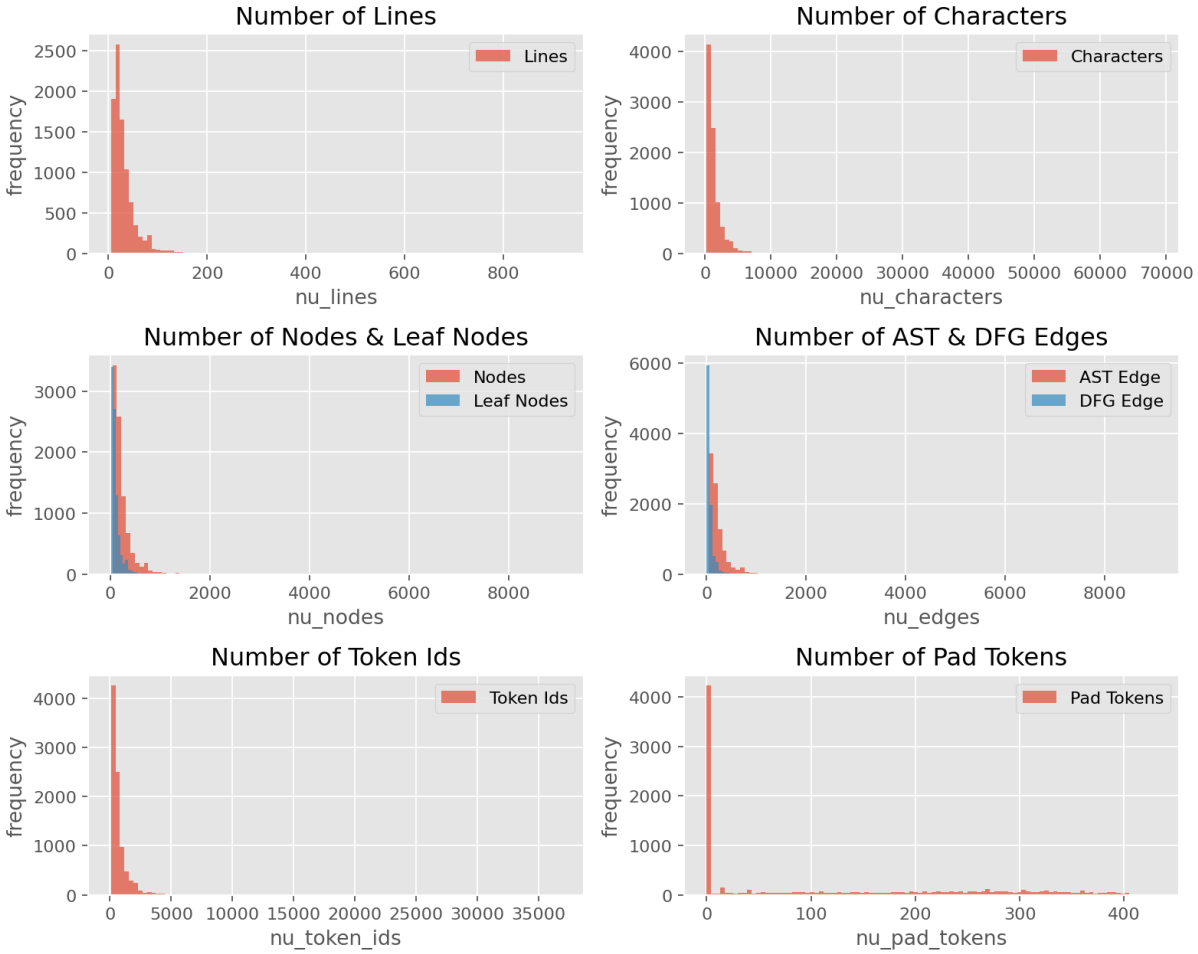
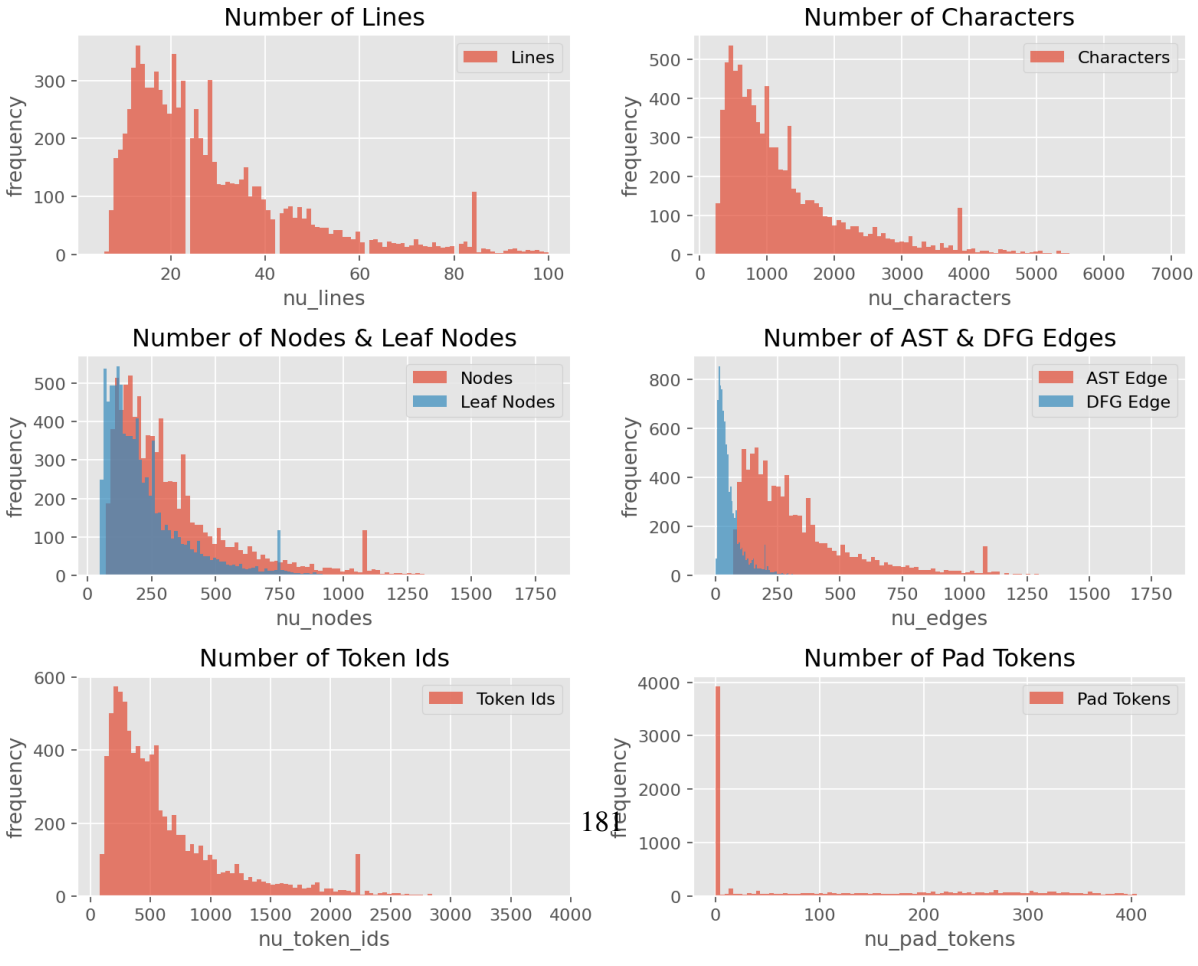


Figure 7: Original

[b]0.32



D.2 Python Data : PoolC

Given in Figure 14 are the original 11 and filtered 13 distributions for Python dataset from PoolC dataset (PoolC, no date).

	nu_lines	nu_characters	nu_nodes	nu_leaf_nodes	nu_ast_edges	nu_dfg_edges	nu_token_ids	nu_pad_tokens
count	9126.0	9126.0	9126.0	9126.0	9126.0	9126.0	9126.0	9126.0
mean	33.9	1579.9	247.6	121.3	246.6	74.9	787.0	113.8
std	40.0	2415.3	327.0	167.0	327.0	153.9	1270.3	133.6
min	5.0	234.0	44.0	16.0	43.0	1.0	81.0	0.0
25%	16.0	605.2	105.0	50.0	104.0	23.0	277.0	0.0
50%	24.0	989.0	169.0	82.0	168.0	42.0	478.0	34.0
75%	38.0	1707.0	271.0	132.0	270.0	77.0	841.0	235.0
max	917.0	68541.0	9041.0	4811.0	9040.0	5981.0	36823.0	431.0

Table 6: BCB original distribution

	nu_lines	nu_characters	nu_nodes	nu_leaf_nodes	nu_ast_edges	nu_dfg_edges	nu_token_ids	nu_pad_tokens
count	2048.0	2048.0	2048.0	2048.0	2048.0	2048.0	2048.0	2048.0
mean	12.2	449.9	76.3	35.7	75.3	14.8	199.7	312.4
std	3.0	107.8	14.3	7.4	14.3	5.5	56.6	56.4
min	5.0	234.0	44.0	16.0	43.0	1.0	81.0	0.0
25%	10.0	369.0	65.0	30.0	64.0	11.0	157.0	274.8
50%	12.0	440.0	76.0	35.0	75.0	15.0	195.0	317.0
75%	14.0	520.0	89.0	41.0	88.0	19.0	237.2	355.0
max	26.0	1500.0	100.0	52.0	99.0	43.0	592.0	431.0

Table 7: BCB filtered distribution

	nu_lines	nu_characters	nu_nodes	nu_leaf_nodes	nu_ast_edges	nu_dfg_edges	nu_token_ids	nu_pad_tokens
count	44950.0	44950.0	44950.0	44950.0	44950.0	44950.0	44950.0	44950.0
mean	19.1	392.4	141.5	69.8	140.5	58.6	213.6	326.6
std	17.5	2061.9	118.9	61.7	118.9	68.2	538.9	147.5
min	1.0	16.0	6.0	2.0	5.0	0.0	8.0	0.0
25%	8.0	137.0	63.0	29.0	62.0	19.0	71.0	251.0
50%	14.0	248.0	105.0	51.0	104.0	38.0	132.0	380.0
75%	24.0	475.0	182.0	90.0	181.0	74.0	261.0	441.0
max	384.0	426657.0	1596.0	846.0	1595.0	2335.0	97574.0	504.0

Table 8: Poolc original distribution

	nu_lines	nu_characters	nu_nodes	nu_leaf_nodes	nu_ast_edges	nu_dfg_edges	nu_token_ids	nu_pad_tokens
count	17570.0	17570.0	17570.0	17570.0	17570.0	17570.0	17570.0	17570.0
mean	9.8	158.5	67.2	31.6	66.2	21.8	83.2	428.8
std	3.8	68.0	18.5	9.6	18.5	10.4	36.5	36.1
min	5.0	33.0	9.0	4.0	8.0	0.0	17.0	0.0
25%	7.0	113.0	53.0	24.0	52.0	14.0	59.0	412.0
50%	9.0	149.0	67.0	32.0	66.0	21.0	77.0	435.0
75%	12.0	193.0	82.0	39.0	81.0	29.0	100.0	453.0
max	61.0	1748.0	100.0	69.0	99.0	69.0	890.0	495.0

Table 9: Poolc filtered distribution

[b]0.3

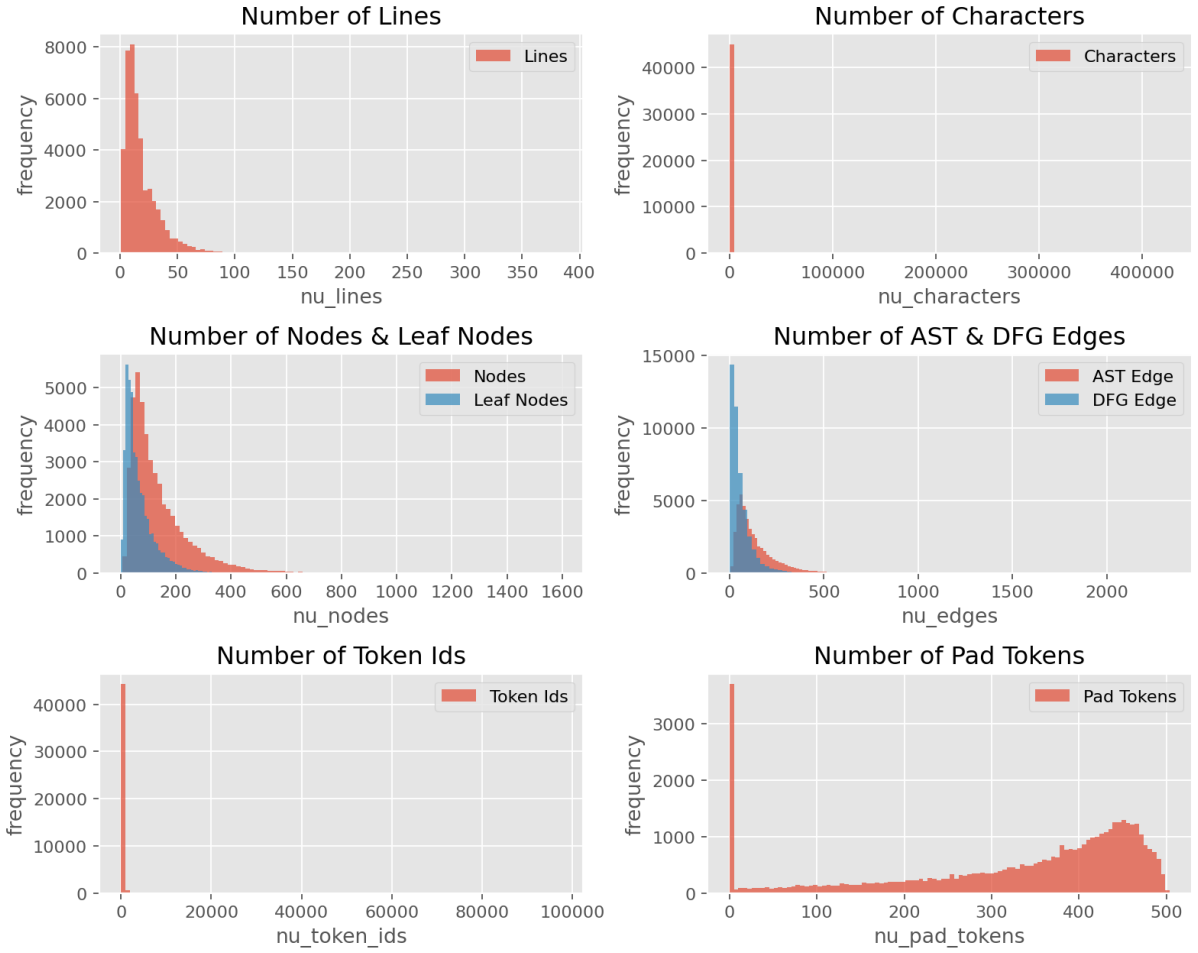
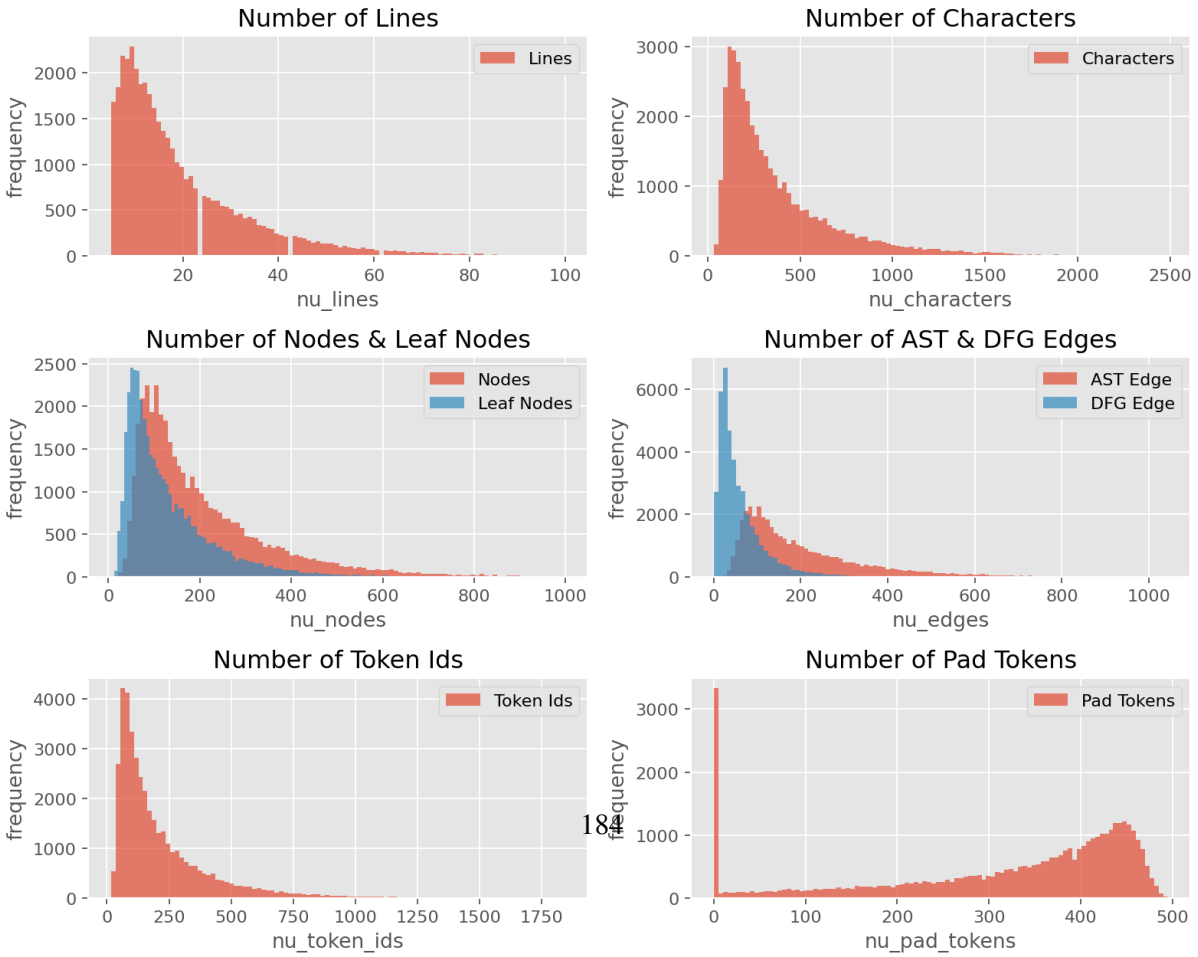


Figure 11: Original

[b]0.3



D.3 Java and Python Data : mix 1

Given in Figure 16 we have the distribution for the mixture of BCB and PoolC dataset. This dataset is made by randomly sampling the filtered datasets of BCB and PoolC examples from each of train, valid, and test splits. This results in total 19K files for source code from both Java and Python together, which results in total 25K Java and 25K python labelled pairs.

[b]0.3

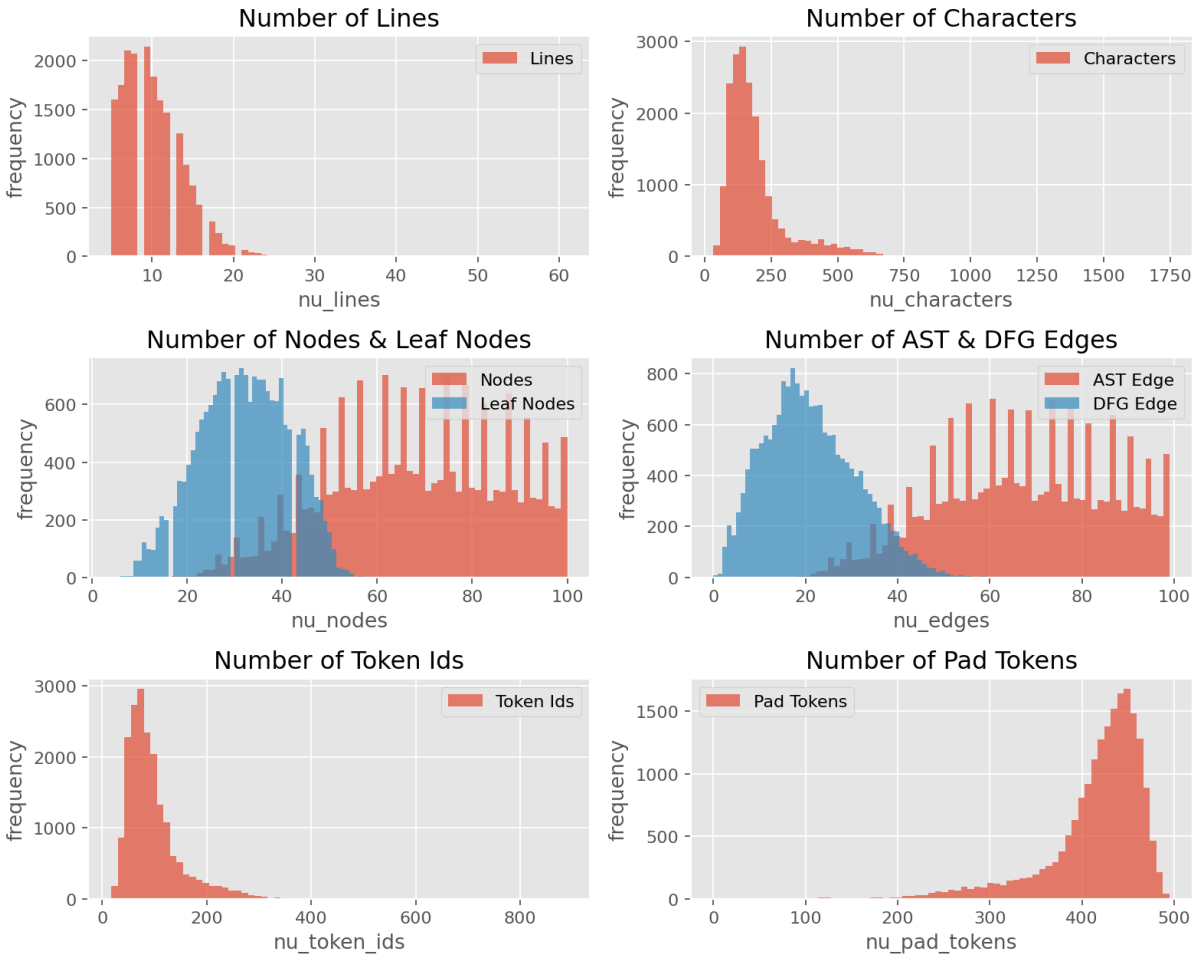


Figure 15: Filtered to Max 100 nodes

Figure 16: Data : Mix 1

	nu_lines	nu_characters	nu_nodes	nu_leaf_nodes	nu_ast_edges	nu_dfg_edges	nu_token_ids	nu_pad_tokens
count	19063.0	19063.0	19063.0	19063.0	19063.0	19063.0	19063.0	19063.0
mean	10.1	189.7	68.2	32.0	67.2	21.0	95.7	416.4
std	3.8	116.3	18.2	9.5	18.2	10.2	53.3	53.0
min	5.0	33.0	9.0	4.0	8.0	0.0	17.0	0.0
25%	7.0	117.0	54.0	25.0	53.0	14.0	61.0	400.0
50%	9.0	157.0	68.0	32.0	67.0	20.0	82.0	430.0
75%	12.0	214.0	83.0	39.0	82.0	28.0	112.0	451.0
max	61.0	1748.0	100.0	69.0	99.0	69.0	890.0	495.0

Table 10: Mix 1 filtered distribution

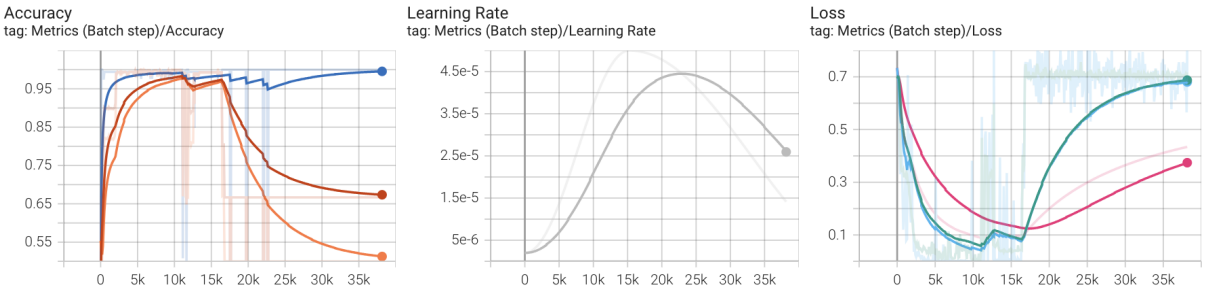
E Training

E.1 Model hyper-parameters

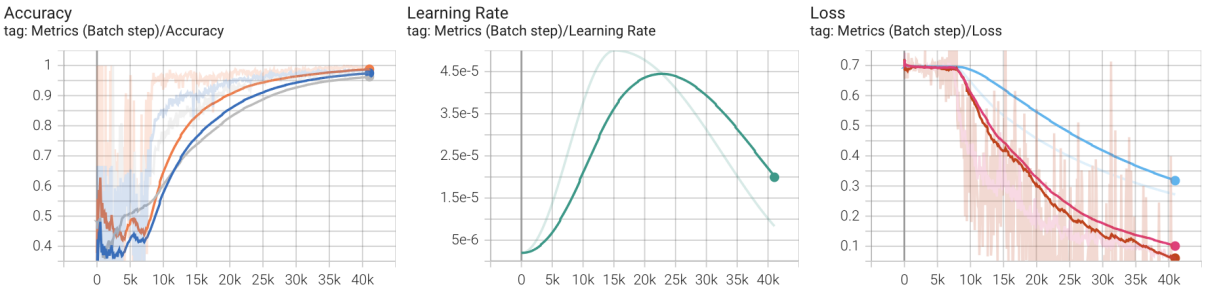
Parameter	CodeBERT	CodeGraph
BATCH_SIZE	16	16
LEARNING_RATE	5e-05	1e-03
OPTIMIZER	AdamW	Adam
SCHEDULER	OneCylceLR	NA
LOSS_FUNCTION	CrossEntropy	FocalLoss
SEQUENCE_LENGTH	512	NA

Table 11: Hyper-parameters of Sequence CodeBERT and Graph CodeGraph models.

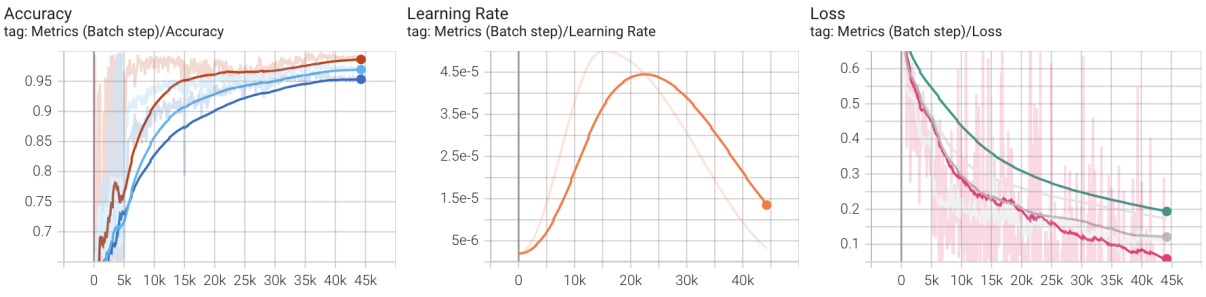
E.2 CodeBERT : Sequence Model



(a) BCB data-set.



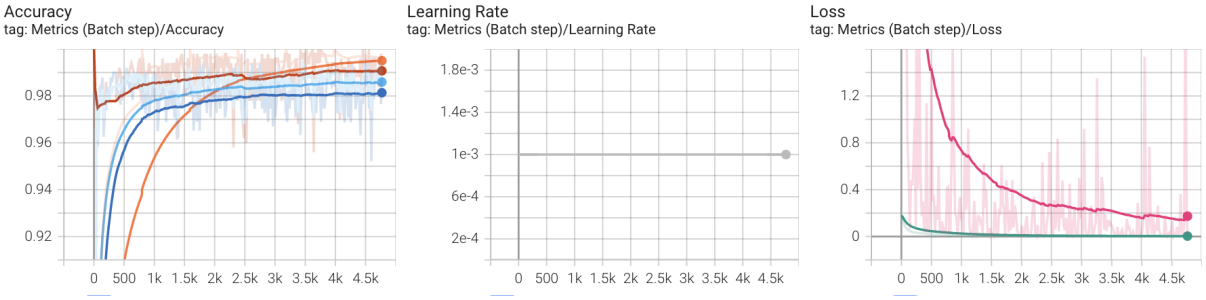
(b) PoolC data-set.



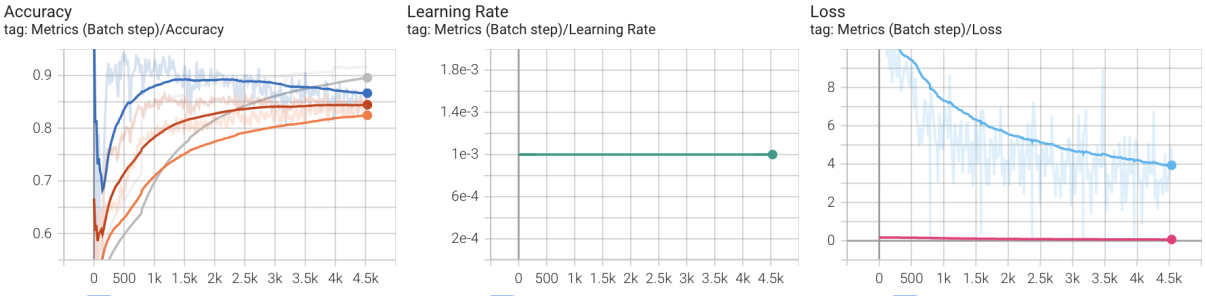
(c) Mix_1 data-set.

Figure 17: Training curves for CodeBERT model

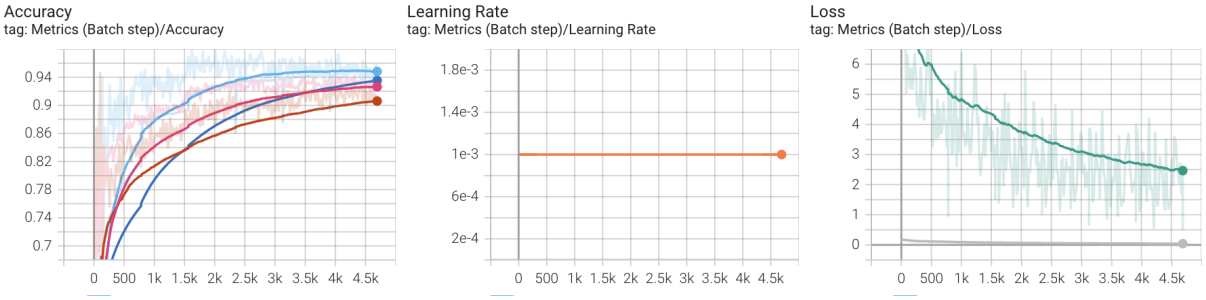
E.3 CodeGraph : Graph Model



(a) BCB data-set.



(b) PoolC data-set.



(c) Mix_1 data-set.

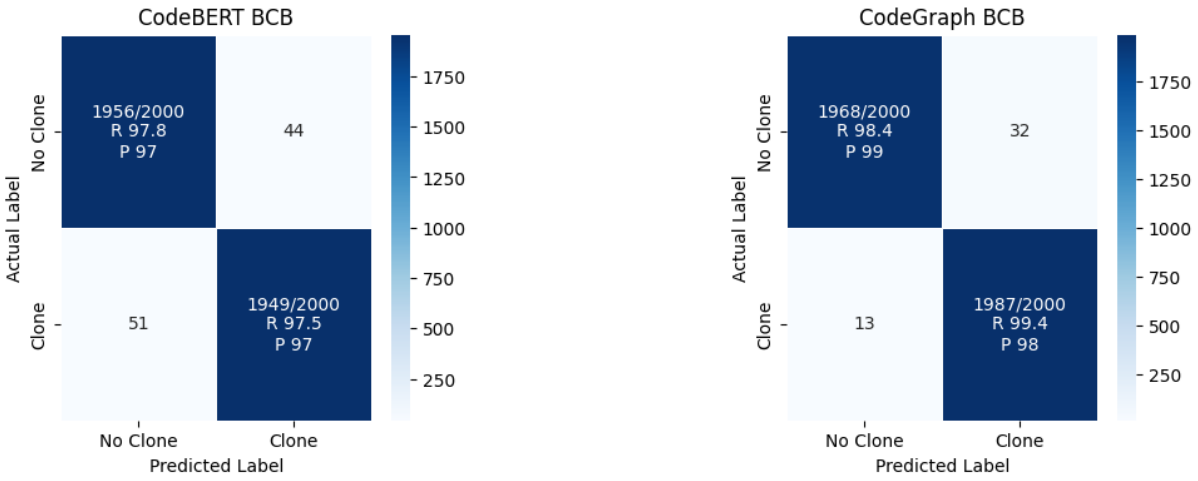
Figure 18: Training curves for CodeGraph model

F Result Analysis

F.1 Confusion Matrix

F.1.1 BCB Dataset

Given in Figure 19 are the confusion matrix for CodeBERT and CodeGraph models.



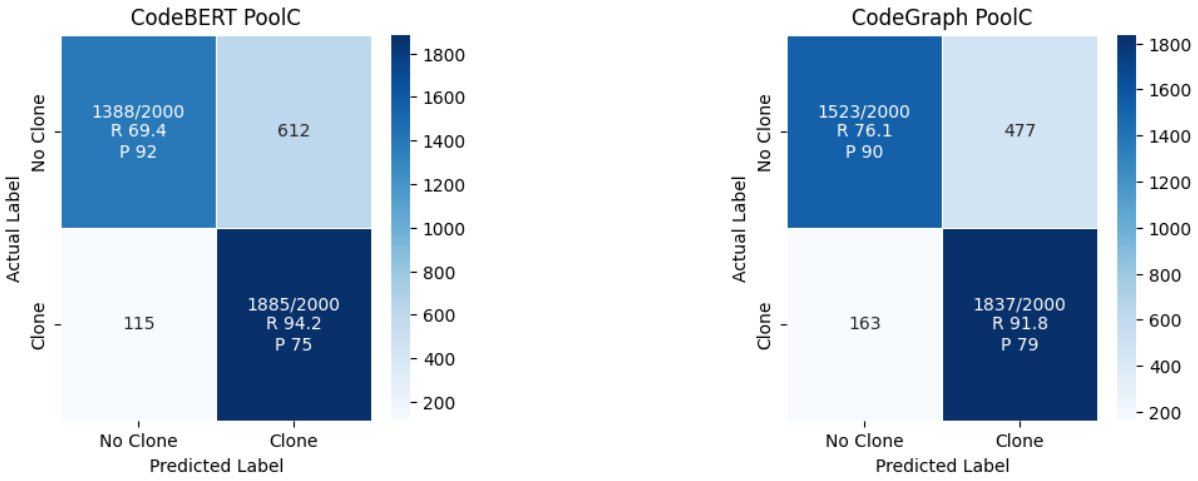
(a) CodeBERT

(b) CodeGraph

Figure 19: Confusion Matrix : BCB

F.1.2 PoolC Dataset

Given in Figure 20 are the confusion matrix for CodeBERT and CodeGraph models.



(a) CodeBERT

(b) CodeGraph

Figure 20: Confusion Matrix : PoolC

F.2 False Positive Analysis

F.2.1 BCB Dataset

- There are 18 False positive from both the models combined. Here we observe that the prediction confidence from CodeBERT is very high above 0.9, where as CodeGraph has prediction confidence on a lower side at 0.5 to 0.6. As observed from Code Pair Listings [3 & 4], [5 & 6]. This suggest that adjusting the classification threshold for CodeGraph can help reduce the False positives which are common in both.

- The False Positives from CodeGraph, but True Negative from CodeBERT is seen to be consistently having less confidence, which is below 0.7. Although these False positives are only predicted by CodeGraph, and CodeBERT very strongly predicts them as True Negative. Examples of Code Pair Listing [7 & 8] following this trend.

- The False Positives from CodeBERT, but True Negative from CodeGraph, is seen to have a consistent prediction with confidence less than 0.9. This can be misleading as this is a higher confidence from CodeBERT. on the same side, CodeGraph doesn't have very strong prediction either, but it is atleast consistently predicting them as TN. Example of Code Pair Listing [9 & 10]

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **Clone(0.91)** | pred_CodeGraph : **Clone(0.62)**

```
public PhoneDurationsImpl(URL url)
    throws IOException {
    BufferedReader reader;
    String line;
    phoneDurations = new HashMap();
    reader = new BufferedReader(new
    InputStreamReader(url.openStream()))
    ;
    line = reader.readLine();
    while (line != null) {
        if (!line.startsWith("***")) {
            parseAndAdd(line);
        }
        line = reader.readLine();
    }
    reader.close();
}
```

Listing 3: code 1

```
public static String getMyGlobalIP() {
    try {
        URL url = new URL(IPSERVER);
        HttpURLConnection con = (
    HttpURLConnection) url.
    openConnection();
        BufferedReader in = new
    BufferedReader(new InputStreamReader
    (con.getInputStream()));
        String ip = in.readLine();
        in.close();
        con.disconnect();
        return ip;
    } catch (Exception e) {
        return null;
    }
}
```

Listing 4: code 2

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **Clone(0.91)** | pred_CodeGraph : **Clone(0.59)**

```
public static LinkedList<String> read(
    URL url) throws IOException {
    LinkedList<String> data = new
    LinkedList<String>();
    HttpURLConnection con = (
    HttpURLConnection) url.
    openConnection();
    BufferedReader br = new
    BufferedReader(new InputStreamReader
    (con.getInputStream()));
    String input = "";
```

```
    while (true) {
        input = br.readLine();
        if (input == null) break;
        data.add(input);
    }
    br.close();
    return data;
}
```

Listing 5: code 1

```
protected Reader getText() throws
    IOException {
    BufferedReader br = new
    BufferedReader(new InputStreamReader
    (url.openStream()));
    String readLine;
    do {
        readLine = br.readLine();
    } while (readLine != null &&
    readLine.indexOf("</table><br clear=
    all>") < 0);
    return br;
}
```

Listing 6: code 2

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **NoClone(0.99)** | pred_CodeGraph : **Clone(0.65)**

```
public PhoneDurationsImpl(URL url)
    throws IOException {

    BufferedReader reader;
    String line;
    phoneDurations = new HashMap();

    reader = new BufferedReader(new
    InputStreamReader(url.openStream()))
    ;
    line = reader.readLine();

    while (line != null) {
        if (!line.startsWith("***")) {
            parseAndAdd(line);
        }

        line = reader.readLine();
    }

    reader.close();
}
```

Listing 7: code 1

```
public void
    alterarQuestaoMultiplaEscolha(
    QuestaoMultiplaEscolha q) throws
    SQLException {
    PreparedStatement stmt = null;
    String sql = "UPDATE
    multipla_escolha SET texto=?,
    gabarito=? WHERE id_questao=?";
    try {
        for (Alternativa alternativa : q
    .getAlternativa()) {
            stmt = conexao.
    prepareStatement(sql);
            stmt.setString(1,
    alternativa.getTexto());
            stmt.setBoolean(2,
    alternativa.getGabarito());
            stmt.setInt(3, q.
    getIdQuestao());
            stmt.executeUpdate();
            conexao.commit();
        }
    } catch (SQLException e) {
        conexao.rollback();
        throw e;
    }
}
```

Listing 8: code 2

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **Clone(0.90)** | pred_CodeGraph : **NoClone(0.67)**

```
public static String getMyGlobalIP() {
    try {
        URL url = new URL(IPSERVER);
        HttpURLConnection con = (
    HttpURLConnection) url.
    openConnection();
        BufferedReader in = new
    BufferedReader(new InputStreamReader
    (con.getInputStream()));
        String ip = in.readLine();
        in.close();
        con.disconnect();
        return ip;
    } catch (Exception e) {
        return null;
    }
}
```

Listing 9: code 1

```
private FTPClient loginToSharedWorkspace
    () throws SocketException,
    IOException {
    FTPClient ftp = new FTPClient();
    ftp.connect(mSwarm.getHost(),
    mSharedWorkspacePort);
    if (!ftp.login(
    SHARED_WORKSPACE_LOGIN_NAME,
    mWorkspacePassword)) {
        throw new IOException("Unable to
     login to shared workspace.");
    }
    ftp.setFileType(FTPClient.
    BINARY_FILE_TYPE);
    return ftp;
}
```

Listing 10: code 2

F.2.2 PoolC Dataset

- There are 344 total False positives predicted by both the models combine. But here we observe that the confidence is following similar trend with BCB dataset, CodeBERT is having higher prediction confidence of Fasle positive, where as CodeGraph has a lower prediction confidence, with max being at 0.71. Here we see that the positive prediction is majorly coming from confusion of syntactically same keywords present in both, for example having extensive usage of if and for loops. This can be seen in the example Code Pair Listing [11 & 12].

- The False positive from CodeGraph, but True Negative from CodeBERT is seen to have consistently lower prediction score from CodeGraph, max being 0.78 and average being 0.58. This again suggests that the CodeGraph is understanding the semantics, and with a high classification threshold, should improve significantly. Here the rational behind why the Graph model seem to get them wrong, is it seems to confused on the syntactic structure of the codes. They might not have same keywords, but the structure syntactically is dominating. A code pair Listing [13 & 14].

- The false positives from CodeBERT, but True negatives from CodeGraph, seems to have stronger prediction of True negatives from Graph models, again showing the Graph models are superior to learn the structural and symatic information with an average score of 0.81. Looking at what might be going wrong with Sequence model would mostly be the keywords having similar names in sequence, but not syntactically similar, as observed in code pair Listing [15 & 16].

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **Clone(0.64)** | pred_CodeGraph : **Clone(0.63)**

```
n=int(input())
x=list(map(int,input().split()))
m=10**15
for i in range(101):
    t=x[:]
    s=sum(list(map(lambda x:(x-i)**2,t))
    )
    m=min(m,s)
print(m)
```

Listing 11: code 1

```
a,b,c,d = map(int, input().split())

ans = -10**18+1
for i in [a,b]:
  for j in [c,d]:
    if ans < i*j: ans = i*j
print(ans)
```

Listing 12: code 2

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **NoClone(0.96)** | pred_CodeGraph : **Clone(0.60)**

```
import sys
x=int(input())

n=1
while(100*n<=x):
    if(x<=105*n):
        print(1)
        sys.exit()
    n+=1
print(0)
```

Listing 13: code 1

```
s = str(input())
t = str(input())
revise = 0
len_str = len(s)
for i in range(len_str):
    if s[i] != t[i]:
        revise += 1
print(int(revise))
```

Listing 14: code 2

Code Pair Example | true_label : **NoClone** | pred_CodeBERT : **Clone(0.83)** | pred_CodeGraph : **NoClone(0.93)**

```
K, N= map(int, input().split())
A = list(map(int, input().split()))
max=K-(A[N-1]-A[0])
for i in range(N-1):
    a=A[i+1]-A[i]
    if max<a:
        max=a
print(K-max)
```

Listing 15: code 1

```
x = float(input())
if 1 >= x >= 0:
    if x == 1:
        print(0)
    elif x == 0:
        print(1)
```

Listing 16: code 2

F.3 False Negative Analysis

F.3.1 BCB Dataset

- There is zero overlap of False Negative between both the models. This is also due to the fact that there are very low false negative overall, due to the model tending to overfit on the dataset.
- The False Negatives from the CodeGraph, but True Positives from CodeBERT, shows consistently lower confidence at an average of 0.7. This shows that we can tweak the classification threshold to handle these lower confidence scores. On further inspection of these cases, which where just 14, shows that this prediction of false negatives are more cause of variation in parameters, which seems to mislead the model to not detect them as clone. Morover we can argue these are Type IV clones, which would be better identified, given more context. An example can be seen in the Code Pair Listing [17 & 18]
- The False Negatives from the CodeBERT, but True Positives from CodeGraph, have a strong very high confidence. This is not helpful, as it is clearly seen that the sequence model is predicting them wrongly as Negataives with a conf average of 0.99. This is a very intersting case, as all the 52 of these cases seems to have the code very different syntactically, but semantically they are same. This shows how the graph model has an edge over the sequence model. This can be seen in the Code Pair Listing [19 & 20]

Code Pair Example | true_label : **Clone** | pred_CodeBERT : **Clone(0.91)** | pred_CodeGraph : **No-Clone(0.58)**

```
public FTPClient sample1c(String server,
     int port, String username, String
    password) throws SocketException,
    IOException {
        FTPClient ftpClient = new
    FTPClient();
        ftpClient.setDefaultPort(port);
        ftpClient.connect(server);
        ftpClient.login(username,
    password);
        return ftpClient;
}
```

Listing 17: code 1

```
public FTPClient sample3b(String
    ftpserver, String proxyserver, int
    proxyport, String username, String
    password) throws SocketException,
    IOException {
        FTPHTTPClient ftpClient = new
    FTPHTTPClient(proxyserver, proxyport
    );
        ftpClient.connect(ftpserver);
        ftpClient.login(username,
    password);
        return ftpClient;
}
```

Listing 18: code 2

Code Pair Example | true_label : **Clone** | pred_CodeBERT : **NoClone(0.99)** | pred_CodeGraph : **Clone(0.97)**

```

public static String getMD5(String s) {

    try {
        MessageDigest m = MessageDigest.
    getInstance("MD5");
        m.update(s.getBytes(), 0, s.
    length());
        s = new BigInteger(1, m.digest()
    ).toString(16);
    }
    catch (NoSuchAlgorithmException ex)
    {
        ex.printStackTrace();
    }
    return s;
}
```

Listing 19: code 1

```
private static byte[] getKey(String
    password) throws
    UnsupportedEncodingException,
    NoSuchAlgorithmException {
     MessageDigest messageDigest =
    MessageDigest.getInstance(Constants.
    HASH_FUNCTION);
     messageDigest.update(password.
    getBytes(Constants.ENCODING));
```

```
    byte[] hashValue = messageDigest.
    digest();
    int keyLengthInbytes = Constants.
    ENCRYPTION_KEY_LENGTH / 8;
    byte[] result = new byte[
    keyLengthInbytes];
    System.arraycopy(hashValue, 0,
    result, 0, keyLengthInbytes);
    return result;
}
```

Listing 20: code 2

F.3.2 PoolC Dataset

- There are 34 False Negatives predicted by both the models combined. Here we see that the average score from Graph model is 0.64 where as 0.87 is the average score from the sequence model. This shows how the sequence model is very confidently wrong, which is not a good sign although, when examples are examined for this case, we find the clone pairs distinctly have a difference in length in the code, which seems to be the reason for these wrong predictions. Code Pair Listing [21 & 22] demonstrates this.
- The False Negatives from CodeGraph, but True positives from CodeBERT, shows a consistently lower score of confidence with average confidence being 0.63. where as the true positives as well from codeBERT have lower confidence average of 0.78. Here the CodeGraph is marignaly doing wrong, and this seems to be due to the code length again. the difference in the size of code snippet is larger. This can be seen again from the Code Pair Listing [23 & 24].
- The False Negatives from CodeBERT, but True Positives from CodeGraph, are around 83. This cases seems to be strongly False at an average score of 0.81 for the CodeBERT, which is not a good sign of the model predicting them wrong confidently, again, the reason looks the same as the snippet sizes being very different. Simialarly average confidence of True positive from CodeGraph is 0.61, which is again not that confident. This can be seen with the Code Pair Listing [25 & 26].

Code Pair Example | true_label : **Clone** | pred_CodeBERT : **NoClone(0.88)** | pred_CodeGraph : **NoClone(0.96)**

```
while True:

    a = input()

    if a == '0':
        break

    print(sum(map(int,*a.split())))
```

Listing 21: code 1

```
n = int(input())
res = 0
while n != 0:
 res = n%10
 dropped = n
 while dropped//10 != 0:
  dropped = dropped//10
  res += dropped%10
 print(res)
 res = 0
 n = int(input())
```

Listing 22: code 2

Code Pair Example | true_label : **Clone** | pred_CodeBERT : **Clone(0.52)** | pred_CodeGraph : **NoClone(0.59)**

```
while True:

    n = input()

    if n == "0":
        break

    print(sum([int(i) for i in n]))
```

Listing 23: code 1

```
while True:
    num = input()
    if int(num) == 0:
        break
    sum = 0
    for i in num:
        a = int(i)
        sum += a
    print(sum)
```

Listing 24: code 2

Code Pair Example | true_label : **Clone** | pred_CodeBERT : **NoClone(0.62)** | pred_CodeGraph : **Clone(0.66)**

```
H,A = map(int,input().split())
cnt = 0
while True:
    if H <= 0:
        print(cnt)
        break
    else:
        H -= A
        cnt += 1
```

Listing 25: code 1

```
h,a = map(int, input().split())
an, bn = divmod(h,a)
if bn == 0:
    print(an)
else:
    print(an+1)
```

Listing 26: code 2

G Discussion

G.1 Limitations

We note the following limitations and concerns in the study:

- The experimental setup data-set size is drastically reduced in order to time-bound the experiments for this research project. Majorly, there are 2 cuts in the data-set size, Firstly, source code files are filtered to only those which have less than or equal to 100 nodes. Secondly, the sample size of the data-set clone and no clone pair is restricted to 50K data-points only. Refer the Appendix D to check the filtering criterion and Section 4.1 to understand the data-point split samples.
- The BCB data-set was significantly reduced after applying the thresholding criterion, resulting in only 2K files out of the original 9K. This led to a over-sampled distribution of the training pairs, which consisted of 50K data-points. As shown in the Results Section 5, this caused the BCB data-set model to over-fit the data, unlike the PoolC data-set model.
- A potential limitation of this study is the discrepancy in the number of trainable model parameters between the sequence model (CodeBERT) and the graph model (CodeGraph). The sequence model has 125M parameters, which is 125 times more than the graph model's 1.1M parameters. This could raise the question of whether the graph model's superior performance is due to its inherent advantages or its lower complexity. However, this also suggests that there is room for further improvement on the graph model by increasing its number of parameters.

G.2 Future Research

Some possible directions for the future research based on the limitations are as follows:

- To help reduce over fitting problem on the Java data-set (i.e. BCB data-set), future research could use samples from other data-sets, like CodeForces (Yeo, 2023), Google Code Jam (Google, no date), which can yield in more diversified data-set for Java language.
- To explore the potential of the graph model (CodeGraph), future research could increase its number of trainable parameters and compare its performance with the sequence model (CodeBERT) under the same complexity level. This would help to determine whether the bigger graph model would still have inherent advantages over the sequence model or not. conversely, one can reduce the parameters on sequence model and check its impact.
- To investigate the effect of batch size on the learning ability of the models, future research could use different batch sizes for both the sequence model (CodeBERT) and the graph model (CodeGraph) and compare their results. This would require a bigger, and / or multiple GPUs. This would help to understand how batch size influences the convergence and stability of these models.
- PoolC data-set false Negatives are majorly due to the code size length differences. This can be solved if trained with longer snippet of codes.
- Qualitiative analysis on examples of experiment 2 to understand the similarity of code-snippets on how multi-lingual settings helped the CodeGraph outperform CodeBERT.

H Data Availability

We have made the data and source code that we used in this paper publicly accessible. Source code is available for replication at: https://github.com/Ataago-AI/clone-detection, and the filtered data-sets can be downloaded from here: https://drive.google.com/drive/folders/1phx8k_JB8HC_HW3nhZLec9BKjRxNCN2b?usp=drive_link