Improving Source Code Similarity Detection Through GraphCodeBERT and Integration of Additional Features

Jorge Martinez-Gil

Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria
jorge.martinez-gil@scch.at

Abstract

This paper presents a novel approach for source code similarity detection that integrates an additional output feature into the classification process to improve model performance. Our approach is based on the GraphCodeBERT model, extended with a custom output feature layer and a concatenation mechanism for improved feature representation. The model was trained and evaluated, achieving promising precision, recall, and f-measure results. The implementation details, including model architecture and training strategies, are discussed. The source code that illustrates our approach can be downloaded from `https://www.github.com/jorge-martinez-gil/graphcodebert-feature-integration`.

Keywords: Feature Integration, GraphCodeBERT, Source Code Similarity

1. Introduction

Accurate and efficient identification of similar source code fragments is essential for ensuring software quality, improving developer productivity, and maintaining code integrity [1, 22, 23]. With the advent of deep learning (DL) and natural language processing (NLP) techniques, transformer-based models have emerged as promising strategies for understanding and processing source code. Recent progress in transformer architectures, particularly models like BERT [3] and its variants, has shown remarkable success in this context.

When pre-trained on large corpora, transformer models can effectively capture semantic and syntactic information, making them highly suitable for source code-related tasks. In this research, we are particularly interested in a transformer variant called GraphCodeBERT [6], specifically designed to manage source code by processing the structural and semantic properties inherent in programming languages.

To improve the capabilities of GraphCodeBERT for source code similarity detection, we propose a novel extension that involves adding a custom output feature layer. This strategy also uses a concatenation mechanism to combine the pooled output from the transformer model with additional processed features. This approach allows the model to represent the source code better since it considers both structural and semantic information and can, therefore, be expected to lead to better results. In this way, the major contributions of this research can be summarized as follows:

- We present a novel approach that extends the capabilities of GraphCodeBERT by integrating additional output features into the sequence classification process. We aim to improve the model's ability to detect source code similarities by providing a richer feature representation.
- We evaluate our model's performance through experiments using well-known datasets. The results show some degree of improvement in precision, recall, and f-measure, validating the effectiveness of our model extension.

The remainder of this paper is organized as follows: Section 2 reviews related work in source code similarity detection and transformer-based models. Section 3 details the methodology, including the model architecture and training strategies. Section 4 presents the experimental setup and results and discusses our findings and their implications. Section 5 concludes the paper and points out potential directions for future research.

2. State-of-the-art

Semantic similarity measurement [19, 13, 14] and in particular, code similarity detection has seen significant advancements over the years, driven by the need to manage and maintain large codebases efficiently. Below, we present the historical evolution and the recent progress that has shaped the current state-of-the-art in this domain.

2.1. Historical Overview

Early approaches to source code similarity detection primarily focused on syntactic analysis, using methods like string matching, token-based comparison, and abstract syntax tree matching [11, 20, 21], as well as source code metrics [7]. While useful [8], these techniques often struggled with coding style and structure variations and faced scalability issues [5], limiting their effectiveness in accurately capturing the semantic similarity between source code fragments. One explored line was the aggregation of basic techniques through ensemble and stacking methods [14, 15], but these showed strong results only on small datasets.

The emergence of machine learning (ML) brought more sophisticated methods [16]. Vector space models and graph-based techniques introduced new ways to represent and compare code fragments [2], incorporating structural properties of code for richer similarity detection [26]. Despite their advancements, ML approaches faced challenges in scaling large datasets and managing diverse programming languages.

DL marked a transformative shift in source code similarity detection [27]. Diverse neural networks were applied to model source code sequences, capturing syntactic and semantic information [25, 28]. These models outperformed traditional methods but struggled to fully capture long-range dependencies and complex code structures.

Transformer-based models, such as BERT and its variants, have significantly impacted NLP and code-related tasks. Pre-trained on extensive datasets, these models have shown exceptional capabilities in understanding context and semantics [9]. The application of transformer models to source code has further evolved with the development of specialized models for programming languages, such as CodeBERT [4] and GraphCodeBERT [6]. These models leverage source code's textual and structural properties, enabling more accurate similarity detection [17]. However, techniques to explain model operations remain an open area of research [12], and more efforts must be made in this direction.

2.2. Contribution Over the State-of-the-art

Our research aims to improve source code similarity detection by extending transformer-based models and feature integration techniques. We extend these advancements by feeding an additional output feature into a transformer-based model to detect code similarity. We aim to improve the model's representation capabilities and classification performance.

Therefore, our approach advances the use of transformer architectures and feature integration for more effective code similarity detection. The primary motivation for this extension is to enrich the representation of source code by incorporating both the structural and semantic properties. While the transformer model effectively captures contextual information, the output feature layer provides additional domain-specific features that the base model might not fully capture.

3. GraphCodeBERT and Additional Feature Integration

GraphCodeBERT is a graph-based pre-trained model based on the transformer architecture for programming languages. It also considers data flow information along with source code sequences. The model is trained on a dataset with several million functions and document pairs for several programming languages. It processes the source code structure for improved understanding and generation. It combines techniques from graph neural networks and transformer-based models like BERT. Below is a mathematical formulation of the key components and processes involved in GraphCodeBERT.

3.1. Problem Statement

Let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ denote the set of source code fragments. The goal is to determine whether a given pair of these code fragments $(C_i, C_j) \in \mathcal{C} \times \mathcal{C}$ are clones, i.e., functionally equivalent, or similar. We frame the clone detection problem as a binary classification task. For a pair of source code fragments (C_i, C_j), the task is to predict the label $y_{ij} \in \{0, 1\}$, where 1 indicates that they are clones and 0 otherwise. We define the input to the classifier as the concatenation of the embeddings of the code pairs:

$$\mathbf{x}_{ij} = [\mathbf{h}_i; \mathbf{h}_j]$$

3.2. Training Objective

The model can be trained using various objectives depending on the task, such as predicting randomly masked tokens in the input sequence or generating summaries for given source code fragments. In the context of this work, the overall loss $\mathcal{L}$ is about matching source code fragments such as the approach presented in [24]. However, our model is trained to minimize the binary cross-entropy loss:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(i,j) \in \mathcal{D}} (y_{ij} \log \hat{y}_{ij} + (1 - y_{ij}) \log(1 - \hat{y}_{ij}))$$

where $\mathcal{D}$ is the training dataset, $\hat{y}_{ij}$ is the predicted probability of C_i and C_j being clones, and θ represents the model parameters.

3.3. Extension of the Model Architecture

The model is based on a transformer architecture, which is extended to include an additional output feature for improved functionality. The primary components of this model are an output feature layer and a classifier. The model processes the input data and generates hidden representations. The output feature layer is a linear layer represented by $\mathbf{W}_1$, which maps the additional output feature to the same dimension as the model's hidden size. The classifier is another linear layer, $\mathbf{W}_2$, which maps the concatenated features to the number of labels required for the classification task.

During the forward pass, given inputs $\mathbf{X} = (\textbf{input\_ids}, \textbf{attention\_mask})$ and an additional output feature $\mathbf{f}_{\text{out}}$, the following computations occur: the model processes the inputs to generate hidden states, represented as $\mathbf{H} = \text{Model}(\textbf{input\_ids}, \textbf{attention\_mask})$. The pooled output is $\mathbf{P}_{\text{pooled}} = \mathbf{H}_{\text{pooler\_output}}$. The additional output feature is processed through the linear layer, resulting in $\mathbf{F}_{\text{processed}} = \mathbf{W}_1 \cdot \mathbf{f}_{\text{out}}$. These two vectors are concatenated to form $\mathbf{C} = [\mathbf{P}_{\text{pooled}}; \mathbf{F}_{\text{processed}}]$. After applying dropout to $\mathbf{C}$, the final logits are computed as $\textbf{logits} = \mathbf{W}_2 \cdot \mathbf{C}_{\text{dropout}}$. For classification tasks, the cross-entropy loss is used to compute the loss:

$\mathcal{L} = \text{CrossEntropyLoss}(\textbf{logits}, \textbf{labels})$. Each data point consists of a pair of code fragments $(\text{code1}, \text{code2})$, a similarity score, and an additional output feature which consists of the execution of the two code fragments and the comparison of their outputs using some semantic textual similarity technique.

4. Empirical Evaluation

In previous work [18], we conducted a state-of-the-art study including unsupervised and supervised strategies. We now proceed to extend the results shown above.

4.1. Experimental Setup

The models have undergone a previous fine-tuning phase, beginning with the loading and randomly splitting a dataset of code fragments into training, validation, and test sets. The approach's core involves training the model to discern source code clone pairs, guided by a trainer configured with specific arguments such as epoch count, batch size, and learning rate adjustments. Finally, performance metrics have been calculated to evaluate the model's effectiveness as the average value of a given number of executions. In our study, we have considered up to ten independent executions.

4.2. Dataset

We use the IR-Plag dataset[1] [10], designed for benchmarking source code similarity techniques in detecting academic plagiarism. The dataset includes 467 code files, with 355 (77%) labeled as plagiarized. It contains 59,201 tokens with 540 unique tokens, offering lexical and compositional diversity. File sizes range from 40 to 286 tokens, averaging 126 tokens per file, making it suitable for studying source code clones.

4.3. Evaluation Criteria

Although accuracy is commonly calculated in studies like this, it is discouraged for unbalanced datasets because it can be misleading; predicting the most frequent class can result in deceptively high accuracy. Therefore, we have chosen to use precision and recall, as this method is more appropriate to separately evaluate false positives (precision) and false negatives (recall). This approach also penalizes models for missing positive instances and making incorrect positive predictions. The f-measure, i.e., the harmonic mean of precision and recall, is then used to rank the effectiveness of different techniques.

[1] `https://github.com/oscarkarnalim/sourcecodeplagiarismdataset`

4.4. Results

Table 1 presents a comparative evaluation of various approaches applied to the IR-Plag dataset. The approaches evaluated include CodeBERT [4], Output Analysis [18], Boosting (XGBoost) [17], Bagging (Random Forest) [17], GraphCodeBERT [6], and our novel variant of GraphCodeBERT. Among the approaches, the novel GraphCodeBERT variant achieved the best performance, with the highest scores in both precision (0.98) and recall (1.00), resulting in an f-measure of 0.99.

Approach	**precision**	**recall**	**f-measure**
CodeBERT [4]	0.72	1.00	0.84
Output Analysis [18]	0.88	0.93	0.90
Boosting (XGBoost) [17]	0.88	0.99	0.93
Bagging (Random Forest) [17]	0.95	0.97	0.96
GraphCodeBERT [6]	0.98	0.95	0.96
Our GraphCodeBERT variant	0.98	1.00	0.99

Table 1: Performance comparison of state-of-the-art techniques on the IR-Plag dataset, evaluated using precision, recall, and f-measure. Our GraphCodeBERT variant outperforms other methods with the highest f-measure of 0.99

4.5. Discussion

Our experimental results reveal several key findings that demonstrate the effectiveness of our approach. These key findings are:

- Firstly, adding an output feature layer has improved the model's performance. Combining the pooled output with the processed output features has enriched the source code representation, leading to better classification results.
- Secondly, the GraphCodeBERT model has demonstrated a strong capability in understanding and representing source code fragments. Its architecture has effectively learned source code similarity, and our custom extension has further improved this capability.
- Lastly, our training and evaluation processes have indicated that the model generalized well to unseen data, achieving high precision, recall, and f-measure scores. This suggests that our approach could be effectively applied to various software engineering tasks that require source code similarity detection to improve the reliability of these applications.

Our results suggest that our approach could improve software maintenance and reduce technical debt. Integrating the additional output feature layer has led to performance improvements, primarily due to the strengthened code understanding capabilities of GraphCodeBERT through our custom extension.

5. Conclusion

In this work, we have extended the GraphCodeBERT model by integrating additional output features to improve classification performance in code similarity detection. Our idea has been to combine transformer-based models with additional features, providing a promising direction for addressing the limitations of earlier methods. Our extended GraphCodeBERT model has significantly improved the process of identifying and classifying similar source code fragments.

Adding an extra output feature layer has combined information from the pooled and processed outputs, resulting in a more detailed representation of the source code. This improvement has increased the model's performance. Our experimental results consistently show that our approach has outperformed the rest of the models regarding precision, recall, and f-measure.

Despite these positive results, future work could be focused on further improvements, such as experimenting with diverse types of additional features, using advanced code augmentation techniques, and applying the model to larger and more diverse datasets. Additionally, integrating the model into real-world applications and conducting user studies should provide valuable information for further refinement and optimization.

Acknowledgments

The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation, and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of SCCH, a center in the COMET - Competence Centers for Excellent Technologies Programme.

References

[1] Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., & Maqbool, B. (2019). A systematic review on code clone detection. *IEEE access*, *7*, 86121–86144.

[2] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*, 1–29.

[3] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, & T. Solorio (Eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (pp. 4171–4186). Association for Computational Linguistics.

[4] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, & Y. Liu (Eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020* (pp. 1536–1547). Association for Computational Linguistics volume EMNLP 2020 of *Findings of ACL*.

[5] Gabel, M., Jiang, L., & Su, Z. (2008). Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering* (pp. 321–330).

[6] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C. B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2021). Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[7] Haque, S., Eberhart, Z., Bansal, A., & McMillan, C. (2022). Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (pp. 36–47).

[8] Hartanto, A. D., Syaputra, A., & Pristyanto, Y. (2019). Best parameter selection of rabin-karp algorithm in detecting document similarity. In *2019 International Conference on Information and Communications Technology (ICOIACT)* (pp. 457–461). IEEE.

[9] Karmakar, A., & Robbes, R. (2021). What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1332–1336). IEEE.

[10] Karnalim, O., Budi, S., Toba, H., & Joy, M. (2019). Source code plagiarism detection in academia with information retrieval: Dataset and the observation. *Informatics in Education*, *18*, 321–344.

[11] Karnalim, O., & Simon (2020). Syntax trees and information retrieval to improve code similarity detection. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (pp. 48–55).

[12] Karnalim, O. et al. (2021). Explanation in code similarity investigation. *IEEE Access*, *9*, 59935–59948.

[13] Martinez-Gil, J. (2014). An overview of textual semantic similarity measures based on web intelligence. *Artif. Intell. Rev.*, *42*, 935–943.

[14] Martinez-Gil, J. (2019). Semantic similarity aggregators for very short textual expressions: a case study on landmarks and points of interest. *J. Intell. Inf. Syst.*, *53*, 361–380.

[15] Martinez-Gil, J. (2022). A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, *10*, 100423.

[16] Martinez-Gil, J. (2023). A comparative study of ensemble techniques based on genetic programming: A case study in semantic similarity assessment. *Int. J. Softw. Eng. Knowl. Eng.*, *33*, 289–312.

[17] Martinez-Gil, J. (2024). Advanced detection of source code clones via an ensemble of unsupervised similarity measures. *CoRR*, *abs/2405.02095*. `arXiv:2405.02095`.

[18] Martinez-Gil, J. (2024). Source code clone detection using unsupervised similarity measures. In P. Bludau, R. Ramler, D. Winkler, & J. Bergsmann (Eds.), *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceedings* (pp. 21–37). Springer volume 505 of *Lecture Notes in Business Information Processing*.

[19] Martinez-Gil, J., & Aldana-Montes, J. F. (2013). Semantic similarity measurement using historical google search patterns. *Inf. Syst. Frontiers*, *15*, 399–410.

[20] Martinez-Gil, J., & Chaves-Gonzalez, J. M. (2019). Automatic design of semantic similarity controllers based on fuzzy logics. *Expert Syst. Appl.*, *131*, 45–59.

[21] Martinez-Gil, J., & Chaves-Gonzalez, J. M. (2021). Semantic similarity controllers: On the trade-off between accuracy and interpretability. *Knowl. Based Syst.*, *234*, 107609.

[22] Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)*, *19*, 1–37.

[23] Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, *74*, 470–495.

[24] Saini, N., Singh, S. et al. (2018). Code clones: Detection and management. *Procedia computer science*, *132*, 718–727.

[25] Wang, W., Li, G., Ma, B., Xia, X., & Jin, Z. (2020). Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 261–271). IEEE.

[26] Wei, H., & Li, M. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI* (pp. 3034–3040).

[27] White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering* (pp. 87–98).

[28] Yu, H., Lam, W., Chen, L., Li, G., Xie, T., & Wang, Q. (2019). Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (pp. 70–80). IEEE.