Записки научных
семинаров ПОМИ
Том 540, 2024 г.

K. Galliamov, L. Khaertdinova, K. Denisova

REFINING JOINT TEXT AND SOURCE CODE EMBEDDINGS FOR RETRIEVAL TASK WITH PARAMETER-EFFICIENT FINE-TUNING

Abstract. Latest developments in natural language processing demonstrate remarkable progress in the code-text retrieval problem. As Transformer-based models used for this task continue to increase in size, the computational costs and time required for end-to-end fine-tuning become substantial. This poses a significant challenge for adapting and utilizing these models when computational resources are limited. Motivated by these concerns, we propose a fine-tuning framework that leverages parameter-efficient fine-tuning (PEFT) techniques. Moreover, we adopt contrastive learning objectives to improve the quality of bimodal representations learned by Transformer-based models. Additionally, for PEFT methods we provide extensive benchmarking, the lack of which has been highlighted as a crucial problem in the literature. Based on extensive experiments with the CodeT5+ model conducted on two datasets, we demonstrate that the proposed fine-tuning framework has the potential to improve code-text retrieval performance by tuning only 0.4% parameters at the most.

§1. Introduction

The advent of Large Language Models (LLMs) based on the Transformer architecture [3] has revolutionized the field of NLP, offering unprecedented capabilities for understanding and generating human-like text. In the domain of software engineering, these advancements have paved the way for the development of tools that can interpret natural language (NL) queries to retrieve the corresponding source code. These tasks hold a significant promise for the development of various programming languages (PLs) for both novice and experienced engineers.

Key words and phrases: Code retrieval, PEFT, CodeT5+, contrastive learning, NLP.

We thank the Innopolis University for providing part of resources and facilities that were essential for conducting the experiments in this work. We extend our sincere gratitude to Professor V. Ivanov from Innopolis University for his invaluable guidance and support throughout our research.

Effective retrieval of source code from NL descriptions, also known as the code search problem, remains challenging, particularly due to the bimodal nature of the task. This problem requires an LLM that can understand and bridge the semantic gap between NL descriptions and PL code. In our work, we emphasize applications of bimodal models [4] and highlight the notable achievements of pre-trained LLMs applied for code retrieval and generation tasks. End-to-end fine-tuning is a commonly used approach to adapt pre-trained models for a specific task. However, in certain scenarios, especially when applied to smaller datasets, this process can become resource-intensive and overparameterized, leading to minimal or negligible improvements in performance [2]. In addition, overfitting becomes a concern, necessitating the implementation of additional strategies to mitigate this issue. The primary goal of this study is to improve the quality of bimodal representations learned by small transformer models in low-resource settings. In particular, we exploit contemporary Parameter-Efficient Fine-Tuning (PEFT) techniques and contrastive learning to reach performance levels achieved by larger models. This can be reformulated as finding a tradeoff between computational load, caused by an extensive number of trainable parameters, and high resulting performance for code retrieval downstream task.

Our approach builds upon the capabilities of the CodeT5+ model [2] with open weights and incorporates a contrastive learning objective to refine the embeddings. Contrastive learning aims to align representations between corresponding text-code instances by maximizing their similarity in the latent space. We evaluate four contemporary PEFT techniques on two datasets with nine PLs during contrastive fine-tuning in low-resource settings. Moreover, we address limitations identified in previous research [1], namely the lack of comprehensive benchmarking for PEFT methods, by providing checkpoints and experiments for our fine-tuned models.

The main contributions of this study include the following.

- We incorporate a contrastive learning objective to refine the embeddings on CodeT5+. This allows us to form relevant NL and PL pairs, along with random negatives, improving the baseline model performance.
- We introduce an open-source framework[1] for fine-tuning Transformer encoders, applying PEFT methods for bimodal retrieval

[1]All final checkpoints can be found in the project repository: `https://github.com/leiluk1/CodeSearcher`https://github.com/leiluk1/CodeSearcher.

tasks. By utilizing PEFT methods, such as LoRA [31], AdaLoRA [32], Prompt-Tuning [34], and (IA)3 [33], we overcome resource limitations and effectively fine-tune the models for each PL. Besides, we address the limitations identified in [1] by providing comprehensive benchmarking for PEFT methods. This includes the provision of checkpoints and benchmarks for fine-tuned models.
- We evaluate a self-assembled dataset as a Proof of Concept (PoC) and the widely used CodeSearchNet (CSN) benchmark [14] to demonstrate the effectiveness of our approach. This provides a comprehensive assessment of our solution's performance across various PLs, including Python, C++, C#, SQL, Javascript, Java, Ruby, Go, and PHP.
- We integrate our fine-tuned checkpoints into the Retrieval-Augmented Generation (RAG) [5] pipeline as codebase documents and query encoder. This results in a 0.5% improvement of the ROUGE score for code generation.

§2. Related Work

2.1. Code-text Retrieval. Efficient source code retrieval has been a major area of research, and multiple methods have been explored to bridge the gap between NL queries and PL code [38]. A crucial aspect of this field is the use of embeddings to represent code in a manner that enables rapid and accurate retrieval based on semantic similarities [15].

In past years, particularly before the advent of Transformer-like architectures, code retrieval approaches relied on a combination of probabilistic models and classical information retrieval approaches [16,17,36,37]. However, in more recent works, significant breakthroughs have occurred leading to advancements in research in this domain. Notably, the introduction of Transformer-based neural architectures, such as CodeBERT [13], has revolutionized the field. CodeBERT, an adaptation of the BERT model specifically for programming languages, has set a benchmark for subsequent models in terms of understanding, generating, and retrieving PL codes. To enhance this approach, GraphCodeBERT [39] introduces data flow during pre-training, which effectively captures a semantic-level structure of code. A comparison of CodeBERT with graph-based embeddings for source code representation was presented in [25].

Another noteworthy approach is the use of Abstract Syntax Trees (ASTs), which provide a structured representation of the code that captures its syntactic features, facilitating more discerning retrieval and search capabilities [43, 44]. UniSBT, for instance, utilizes syntax-aware embeddings derived from ASTs to enhance the relevance of the retrieved code snippets [42].

CodeT5+ extends the T5 model to handle code intricacies, offering improvements in both code understanding, retrieval, and generation tasks [2]. Finally, OpenAI introduced large cpt-code models (from 0.3 to 175 billion parameters) that are pre-trained from scratch using contrastive learning [40]. While these models have shown groundbreaking performance on code retrieval tasks, it is important to acknowledge that fully pre-training such models with the use of large batch sizes necessitates enormous computational resources and time.

Despite these advancements, the field continues to evolve, with ongoing research seeking to refine these models further and address the challenges posed by the diversity of programming languages and the complexity of code semantics. The work presented in this paper builds upon these foundational efforts, aiming to refine metalanguage embeddings for the retrieval task in low-resource settings through PEFT methods and contrastive learning.

2.2. Parameter Efficient Fine-Tuning. In recent years, the number of parameters in Transformer-based models used in NLP has been growing from millions to trillions. Thus, fine-tuning the parameters of such large models requires substantial computational resources and an enormous amount of time. To overcome these challenges, specific techniques referred to as PEFT have been introduced. In general, these methods allow training a relatively small number of additional parameters in the model, thereby striking a balance between fine-tuning under limited resource scenarios and enabling effective learning of task-specific parameters [1, 29]. For instance, in the context of automated code generation, the work [30] investigates the application of PEFT methods.

Prompt tuning is one such approach that introduces additional learnable prompt tokens into the model input. During fine-tuning, only the prompt parameters are updated, whereas the pre-trained parameters remain fixed [34]. Additionally, Wang et al. [35] have applied prompt tuning to code-related tasks and demonstrated its superiority over fine-tuning models like CodeT5 [41] and CodeBERT [13] in various tasks including code summarization and code translation. Another notable technique, Infused Adapter ((IA)3) [33] utilizes learned vectors to scale activations, introducing a small number of extra parameters. Additionally, the work [31] presents Low-Rank Adaptation (LoRA) that introduces two trainable low-rank matrices for weight update. Adaptive Low-Rank Adaptation (AdaLoRA) [32] extends the technique by dynamically adjusting the rank of the matrices to control the allocation budget.

In this work, we explore the usage of all PEFT techniques described above in combination with contrastive learning to address the source code retrieval task.

2.3. Contrastive Learning. Over the last decade, numerous self-supervised learning (SSL) methods have been proposed to learn deep representations without using annotated data [18]. One of the techniques that has shown state-of-the-art performance in various research fields is contrastive learning. Contrastive learning has been extensively utilized to align representations between different modalities and views in the related literature [19, 20, 22, 28]. The main idea behind this family of approaches is to maximize the alignment between semantically similar instances by contrasting them against dissimilar ones [21]. Specifically, a common way to formulate a contrastive learning objective is to group instances into pairs, positive and negative, and maximize similarities between positive ones. In these settings, a positive pair is formed by different views or modalities corresponding to the same instance.

Initially, contrastive learning has been suggested as an SSL pre-training strategy for deep neural networks. Nevertheless, recent literature presents frameworks that incorporate contrastive losses to fine-tune large models on certain tasks, including information retrieval. For instance, Pour and Farinneya et al. [23] exploited contrastive learning to fine-tune BERT embeddings for retrieving relevant items based on their reviews. In [24], a similar idea, based on CLIP architecture [22], has been proposed for video-text retrieval tasks. Building upon these findings, our study explores fine-tuning the CodeT5+ model [2] using contrastive learning for source code retrieval given textual descriptions.

§3. Methodology

3.1. Fine-tuning Approach. The proposed fine-tuning framework, depicted in Fig. 1, utilizes the contrastive learning objective and Parameter-Efficient Fine-tuning techniques. Specifically, we propose to align representations of matching code-text pairs in a joint feature space by utilizing

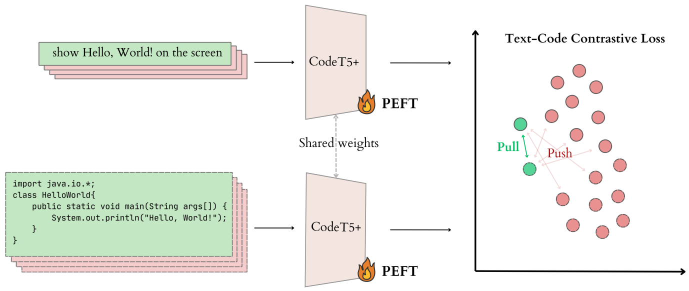

Figure 1. The proposed fine-tuning framework. Contrastive loss aims to maximize similarities between corresponding code-text pairs and minimize the similarities of non-matching pairs. For visual clarity, that is schematically demonstrated for one positive pair in a batch, namely "hello world" text and Java code pair. During fine-tuning, CodeT5+ is tuned using PEFT techniques.

contrastive loss, which is further elaborated on in Section 3.2. Furthermore, our contrastive learning approach fine-tunes the CodeT5+ embedding model (110 million parameters) using PEFT methods, that involve training a small proportion of parameters through back-propagation (Section 3.3). An important detail of our fine-tuning is that it is performed for each programming language separately, thus introducing PEFT parameters that adapt the general-purpose CodeT5+ representations for the code retrieval task on a certain programming language.

3.2. Contrastive Learning to Align Code-text Pairs. In the pursuit of refining source code retrieval from NL descriptions, our methodology emphasizes the alignment of code-text pairs as a critical aspect. This alignment process is instrumental in bridging the semantic gap between NL queries and PL codes, which is essential for effective retrieval systems. In order to achieve this, we have employed PEFT methods on the CodeT5+ model, leveraging its bimodal capabilities to understand and generate code. The training of embeddings differs from generating a PL conditioned on NL input. More specifically, the objective would not be next-token likelihood optimization, but rather bimodal contrastive loss. Contrastive learning is an approach that aims to bring closer the embeddings of relevant NL and PL pairs while pushing apart those of irrelevant pairs. This is achieved by using a text-code contrastive loss function during training. Consequently, the model acquires the ability to generate more discriminative embeddings, thereby facilitating improved alignment between NL queries and their corresponding code.

In our study, contrastive loss will pull together embeddings for positive (relevant) samples and pull apart irrelevant pairs [2]. Formally, given a mini-batch of N code-text pairs $\{\boldsymbol{h}_i^c, \boldsymbol{h}_i^t\}_{i=1}^N$ as normalized lower-dimensional representations, relevant pairs of code vector $\boldsymbol{h}_i^c$ and text vector $\boldsymbol{h}_i^t$ are considered positive as they correspond to the same i-th instance in a batch. The loss $l_i^{t \to c}$ treating textual representation $\boldsymbol{h}_i^t$ from i-th example as an anchor can be computed as follows:

$$l_i^{c \to t} = -\log \frac{\delta(\boldsymbol{h}_i^c, \boldsymbol{h}_i^t)}{\sum_{k=1}^N \delta(\boldsymbol{h}_i^c, \boldsymbol{h}_k^t)}, \tag{1}$$

where $\delta(\boldsymbol{h}_i^c \boldsymbol{h}_i^t) = \exp\left(\frac{\boldsymbol{h}_i^{cT} \boldsymbol{h}_i^t}{\tau}\right)$. Therefore, the total loss aggregated for the whole batch of views c and t can be averaged as

$$L^{c,t} = \frac{1}{2N} \sum_{i=1}^{N} (l_i^{c \to t} + l_i^{t \to c}) \tag{2}$$

3.3. Trainable Parameters. We utilize PEFT methods that involve training a small proportion of newly added parameters through back-propagation. In our setup, during training, all model weights are frozen except for the ones added by PEFT methods. These methods are defined as follows:

- LoRA [31]: introducing low-rank addends to Q, V tensors of Attention.
- AdaLoRA [32]: LoRA-based approach with addends ranks changing during training.
- (IA)3 [33]: learnable scaling vectors for Q, K, and linear tensors.
- Prompt-Tuning [34]: learnable vectors prepended to hidden input representation.

§4. Experimental setup

For this experimental study, we have designed a comprehensive experimental setup to evaluate the efficacy of PEFT methods applied to the CodeT5+ model for the task of source code retrieval for small models.

Table 1. Size of the CodeSearchNet dataset split into programming languages and train/validation/test subsets.

	Ruby	**JS**	**Go**	**Java**	**PHP**
train	48K	123K	317K	454K	523K
valid	2.2K	8.2K	14K	15K	26K
test	2.2K	6.4K	14K	26K	28K

Our experiments were conducted with a focus on optimizing the alignment of NL and PL embeddings, which is crucial for the successful retrieval of source code corresponding to NL descriptions.

4.1. Data Selection. For our fine-tuning framework, we utilized two distinct datasets: the well-established CodeSearchNet benchmark and a custom dataset assembled specifically for this study.

4.1.1. *CodeSearchNet.* The CodeSearchNet (CSN) [14] benchmark is a large-scale dataset that has been widely used in the field for evaluating code retrieval models. This dataset encompasses a diverse range of programming languages, including Java, JavaScript, Go, PHP, and Ruby. This dataset provides a comprehensive set of NL documentation and PL code pairs. Table 1 demonstrates dataset split sises for each PL. The CSN dataset has been used as a benchmark for demonstrating the robustness and effectiveness of our approach across a wide range of programming languages, allowing us to assess the performance of our fine-tuned models comprehensively.

4.1.2. *Custom Dataset.* To prove our concept and demonstrate the potential of PEFT methods in enhancing the CodeT5+ model's ability to map NL descriptions to source code we collected a relatively small dataset compared to the CodeSearchNet. The custom dataset includes a collection of NL text and PL code pairs, with a particular emphasis on smaller and less represented programming languages, which are neglected in larger benchmarks.

In our experimental setup, we investigated various PL datasets, including Python [7, 9, 10], C# [7], C++ [7], SQL [8], Solidity [11], and Assembly [12]. However, we determined that the Solidity and Assembly datasets were not suitable, as it was impossible to match PL and NL pairs: Assembly dataset did not contain documentation, and Solidity

Table 2. Size of the custom dataset per programming language and split.

	Python	**C#**	**C++**	**SQL**
train	10M	58K	63K	62.9K
valid	120K	3K	3K	6.9K
test	123K	5.4K	5.6K	7.7K

dataset was not publicly available anymore. We therefore excluded these datasets. During the data exploration, a thorough analysis of the selected datasets was conducted to gain insight into their characteristics and identify any potential biases or data quality issues. Several datasets, particularly Search4Code [8], were removed, as they do not provide any code snippets and NL query required for the task of code search. Moreover, pairs with non-English natural language queries were removed. Additionally, we deleted NL and PL pairs, where each sample in the pair was too short, specifically less than three tokens, or too long, the boundary was defined as a hyperparameter. The absence of appropriate code snippets and related code descriptions restricts our ability to include them in the final dataset.

Following the exploration of these datasets, we merged them according to their respective programming languages. Then, tokenization and common preprocessing steps were performed for both the NL description and PL code. For further model fine-tuning, we establish the token lengths of NL text and PL code. Figure 2 demonstrates detailed information on the distribution of both the PL and NL token lengths in the dataset.

As a result, we obtained the dataset with sample split sizes demonstrated in Table 2. By incorporating these datasets into our experiments, we aimed to showcase the versatility of our fine-tuning framework and its applicability to a wide array of programming languages.

4.2. Fine-tuning Details. For the base pre-trained model we have taken a CodeT5+ encoder[2] with a lower dimension projection and normalization head on top. The above-mentioned encoder comes from the CodeT5 model,

[2]https://huggingface.co/Salesforce/codet5p-110m-embedding

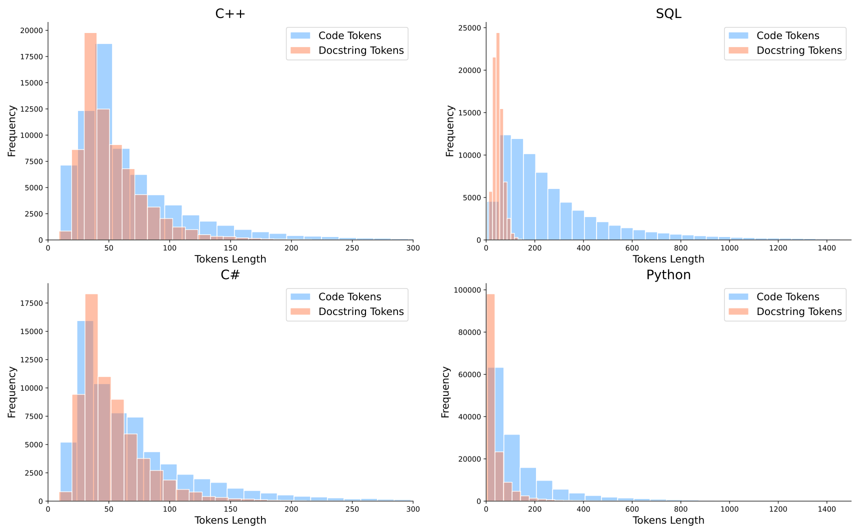

Figure 2. The distribution of token length for NL code docstring and PL code snippets, respectively, for PLs included in our dataset.

Table 3. Tunable parameters for PEFT methods. Prompt-tuning was done with 10 tokens by default.

	AdaLoRA	**LoRA**	$(IA)3$	**Prompt**
tunable %	0.402	0.268	0.025	0.007
tunable #	442,656	294,912	27,648	7680

pre-trained on large-scale datasets on tasks of code generation, span denoising, contrastive objectives, and others. For more details on how the base model was pre-trained, one can refer to [41].

For fine-tuning, we chose to train on pairs with at most 256 NL and 256 PL maximum-length tokens. For the embedding loss, the initial temperature τ was set to 0.08. The learning rates were set to 0.001, batch size was set to 128 and the gradient accumulation steps were set to 4 in all the experiments. Furthermore, a Cosine Annealing Scheduler was used.

When tuning the embeddings, we used Prompt-tuning instead of Prompt Encoder from the PEFT library. This decision was made because the latter failed to overfit a single batch, and Prefix-Tuning was not supported for the embedding extraction task.

All final checkpoints may be found in our project repository[3].

4.3. Evaluation Methods. We use Mean Reciprocal Rank (MRR) as the evaluation metric. For the evaluation part of the validation and test sets, we have utilized two approaches to calculate MRR. The first approach, adopted from [13], involves computing similarities for all possible NL, and PL pairs in the test dataset. The MRR is then calculated only on the ranks that are not greater than 1000. The second approach, proposed in [14], splits the test dataset into chucks of 1000 pairs each (if the last chunk is smaller, then it is discarded). Regular mean reciprocal ranks are computed for each of these chucks, and these MRRs are then averaged. In each table with the evaluation part presented in our work, the chosen method for calculating the MRR is given in the corresponding description.

4.4. Further Usage with RAG. One of the points of application of our models could be Retrieval-Augmented Generation (RAG) [5]. We incorporate our model as an embedding model for both text and code chunks. The retrieval database is built on code samples, and corresponding docstrings are used as queries. For reader LLM, we have chosen "deepseek-coder-6.7b-instruct" from [26]. To provide a visual representation of our approach in the context of RAG, please refer to Figure 3. This resulted in a minor but stable improvement of the ROUGE metric compared to the non-tuned baseline model. We provide more details on evaluation results in Section 5.3.

§5. Results

5.1. Custom Dataset. First, the proposed fine-tuning strategy was applied to the custom dataset described in Section 4.1. In particular, we evaluated the suggested contrastive learning objective along with PEFT techniques on each programming language presented in the dataset. Table reftable:mrr summarizes the MRR scores obtained on test sets of the dataset. In this table, we also show the baseline results that correspond to the pre-trained CodeT5+ without any further fine-tuning. According to

[3]https://github.com/leiluk1/CodeSearcher/tree/main/checkpoints

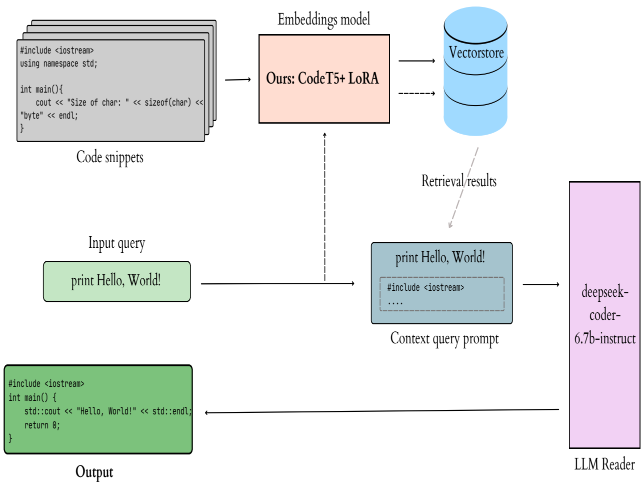

Figure 3. Integration of the best checkpoints of our fine-tuned models into the RAG pipeline for different PLs used in the study. The figure provides an example of the code generation for C++.

the obtained results, all the utilized PEFT techniques lead to an increase in performance. Among these methods, AdaLoRA demonstrates the highest scores in all four programming languages. Specifically, it boosts the performance by about 17% on C++ and C#, more than 6% on SQL, and approximately 9% on Python compared to the baseline scores.

Furthermore, Fig. 4 illustrates the validation losses during fine-tuning. Based on the obtained curves a few observations can be made. First, reparametrization-based methods such as LoRA and AdaLoRA seem to have the fastest convergence. (IA)3 converges a little bit slower, while Prompt Tuning is the longest to converge. What is more, for SQL, the loss converges to higher values. This can be explained by the fact that SQL was not used during the pre-training of CodeT5+ [2], meaning that the model performs in zero-shot settings. Nevertheless, our fine-tuning improves the performance of this language as well, as highlighted above.

Finally, we note that due to limited computational resources and the large size of Python data, we conducted fine-tuning for only 3 epochs.

Table 4. Evaluation of embedding models: MRR on test datasets (for Python, the number of test pairs was limited to 32k) computed in the same way as in CodeBERT [13].

	Python	**C#**	**C++**	**SQL**
Baseline	75.89	22.15	21.69	06.15
LoRA	83.22	38.75	39.23	12.57
AdaLoRA	**83.95**	**39.19**	**39.31**	**12.80**
$(IA)3$	78.48	32.99	32.61	08.97
Prompt	81.33	28.85	23.84	7.91

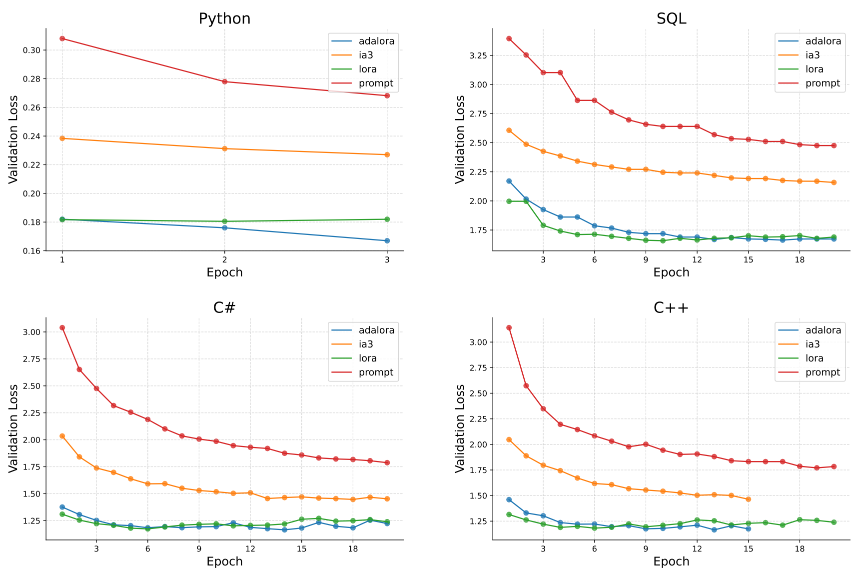

Figure 4. Validation losses plots for the embeddings model on PoC datasets.

However, as mentioned earlier, we still observe a substantial enhancement in performance after our fine-tuning. We leave the exploration of more thorough fine-tuning using Python data for future work.

Table 5. Embeddings model evaluation results on CSN benchmark. For MRR calculation, an approach from [14] was used.

	Ruby	JS	Go	Java	PHP
Baseline	76.11	74.42	77.69	75.66	77.39
LoRA	77.74	**76.99**	**79.10**	**77.82**	**80.43**
AdaLoRA	**77.88**	76.76	78.99	77.53	80.24
$(IA)3$	76.98	76.14	78.95	77.37	79.70
Prompt	74.53	74.93	78.49	76.73	79.00

5.2. Benchmarking on CSN. We evaluated our approach on the CSN benchmark described in Section 4.1. The results, provided in Table 5, demonstrate the MRR scores achieved on the test sets of respective PLs within the CSN dataset. Similarly, Table 6 provides a comparison of MRR scores obtained in our approach with those of SOTA models, while also considering the number of trainable parameters.

Based on results from Table 5, we observed the greatest improvement over baseline performance when using LoRA and AdaLoRA methods. However, compared to the results on the custom dataset, LoRA slightly outperforms AdaLoRA almost for all programming languages, except Ruby. Selected PEFT methods outperformed the baseline across all programming languages presented in CSN. LoRA achieved the highest MRR JavaScript (+2.5%), Go (+1.3%), Java (+2.1%), and PHP (+3%) followed closely by AdaLoRA, which performed the best results on Ruby (+1.7%). IA3 also showed improvements over the baseline but was less effective than LoRA and AdaLoRA.

The findings presented in Table 6 highlight the performance of our approach in comparison to SOTA models. Our approach stands out as achieving the second-best results, following the performance of cpt-code models. It is important to note that the cpt-code S and M models, which outperform our approach, are significantly larger with 0.3 and 1.2 billion parameters, respectively. Moreover, all these parameters were tuned during pre-training using contrastive learning in an end-to-end fashion. In addition, we achieve an increase of 6.7% in average MRR compared to BERT-based embedders. A notable aspect of our approach is the fine-tuning of CodeT5+ using LoRA, which not only contributes to its high

Table 6. Comparison of our approach against state-of-the-art models. We underline the second-best results after cpt-code models. We take the text-to-code retrieval results for other models, except ours and CodeT5+ baseline, from [2] and [40]. For the Go benchmark, we obtained substantially lower results compared to the original CodeT5+ paper [2] when using the open-source checkpoint for CodeT5+.

Model	**CodeSearchNet**					**Trainable**
	Ruby	**JS**	**Go**	**Java**	**PHP**	**parameters**
GraphCodeBERT [39]	70.3	64.4	<u>89.7</u>	69.1	64.9	125M
CodeBERT [13]	67.9	62.0	88.2	67.6	62.8	125M
Ours: CodeT5+ baseline	76.1	74.4	77.7	75.7	77.4	220M
Ours: CodeT5+ AdaLoRA	<u>77.9</u>	76.8	79.0	77.5	80.2	443k
Ours: CodeT5+ LoRA	77.7	<u>77.0</u>	79.1	<u>77.8</u>	<u>80.4</u>	295k
cpt-code S [40]	**86.3**	86.0	**97.7**	94.0	96.7	300M
cpt-code M [40]	85.5	**86.5**	97.5	**94.4**	**97.2**	1.2B

performance but also makes it the most advantageous option in terms of computational expenses. This technique finds an optimal balance between the number of trainable parameters and MRR scores, further bridging a gap between CodeT5+ and cpt-code.

5.3. RAG Case Study. Exact settings for our RAG setup can be found in Section 4.4. Computing the ROUGE metric [27] over 1000 docstring queries from CSN test split has given an 0.5% increase in ROUGE-L, 0.6% increase in ROUGE-2 and 0.45% increase in ROUGE-1. It is definite that further experimentation with various reader models and prompts is needed, which lies outside of the scope of our work.

§6. Conclusion

In this paper, we adopted a contrastive learning objective to enhance source code embeddings for retrieval tasks along with fine-tuning CodeT5+ with PEFT methods in low-resource settings. To address the existing limitation of comprehensive benchmarks for PEFT techniques, we developed an open-source framework for fine-tuning CodeT5+ using PEFT techniques. We evaluated our fine-tuned model on diverse programming language datasets, including our custom dataset and the CodeSearchNet.

Our findings provide a foundation for future studies and highlight the potential of PEFT techniques. However, it is crucial to acknowledge the limitations of our work. In particular, this is related to limited resource power, the small batch size used in fine-tuning using contrastive learning, and the small number of epochs used for large datasets.

In future work, we propose exploring additional PEFT methods, expanding the evaluation on different code datasets, and investigating techniques to handle larger codebases. Another direction of research could be aligning embedder models for specific reader models in the RAG pipeline.

References

1. V. Lialin, V. Deshpande, and A. Rumshisky, *Scaling down to scale up: A guide to parameter-efficient fine-tuning*, arXiv preprint arXiv:2303.15647 (2023).
2. Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, *CodeT5+: Open Code Large Language Models for Code Understanding and Generation.* — Proc. 2023 Conf. Empir. Methods Nat. Lang. Process. (2023), pp. 1069–1088.
3. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need.* — Adv. Neural Inf. Process. Syst. **30** (2017).
4. P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, *Supervised contrastive learning.* — Adv. Neural Inf. Process. Syst. **33** (2020), pp. 18661–18673.
5. P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, arXiv preprint arXiv:2005.11401 (2021).
6. M. Hasan, T. Muttaqueen, A.A. Ishtiaq, K.S. Mehrab, M.M.A. Haque, T. Hasan, W.U. Ahmad, A. Iqbal, and R. Shahriyar, *CoDesc: A Large Code–Description Parallel Dataset.* — Findings Assoc. Comput. Linguist.: EMNLP (2021).
7. M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C.K. Reddy, *Xlcost: A benchmark dataset for cross-lingual code intelligence*, arXiv preprint arXiv:2206.08474 (2022).
8. N. Rao, C. Bansal, and J. Guan, *Search4Code: Code search intent classification using weak supervision.* — Proc. 2021 IEEE/ACM 18th Int. Conf. Min. Softw. Repositories (2021), pp. 575–579.
9. H. Yao et al., *StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow.* — Proc. World Wide Web Conf. (2018), pp. 135–144.
10. M. Bahrami, N.C. Shrikanth, S. Ruangwan, L. Liu, Y. Mizobuchi, M. Fukuyori, W.-P. Chen, K. Munakata, and T. Menzies, *Pytorrent: A python library corpus for large-scale language models*, arXiv preprint arXiv:2110.01710 (2021).

11. C. Shi, Y. Xiang, J. Yu, and L. Gao, *Semantic code search for smart contracts*, arXiv preprint arXiv:2111.14139 (2021).
12. S. Kairajärvi, A. Costin, and T. Hämäläinen, *ISAdetect: Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code*. — Proc. Tenth ACM Conf. Data Appl. Secur. Priv. (2020).
13. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., *CodeBERT: A pre-trained model for programming and natural languages*, arXiv preprint arXiv:2002.08155 (2020).
14. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, *CodeSearchNet challenge: Evaluating the state of semantic code search*, arXiv preprint arXiv:1909.09436 (2019).
15. Y. Xie, J. Lin, H. Dong, L. Zhang, and Z. Wu, *Survey of code search based on deep learning*. — ACM Trans. Softw. Eng. Methodol. **33**(2) (2023), pp. 1–42.
16. S. Chatterjee, S. Juvekar, and K. Sen, *Sniff: A search engine for Java using free-form queries*. — Fundam. Approaches Softw. Eng. Int. Conf. (2009), pp. 385–400.
17. E. Hill, M. Roldan-Vega, J.A. Fails, and G. Mallet, *NL-based query refinement and contextualized code search results: A user study*, Proc. 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) (2014), 34–43.
18. X. Liu, F. Zhang, Z. Hou, L. Mian, Z. Wang, J. Zhang, and J. Tang, *Self-supervised learning: Generative or contrastive*. — IEEE Trans. Knowl. Data Eng. **35**(1) (2021), pp. 857–876.
19. R. Brinzea, B. Khaertdinov, and S. Asteriadis, *Contrastive learning with cross-modal knowledge mining for multimodal human activity recognition*, Proc. 2022 International Joint Conference on Neural Networks (IJCNN) (2022), pp. 1–8.
20. Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen, *Graph contrastive learning with augmentations*. — Advances in Neural Information Processing Systems **33** (2020), pp. 5812–5823.
21. T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, *A simple framework for contrastive learning of visual representations*, Proc. Int. Conf. Mach. Learn. (2020), pp. 1597–1607.
22. A. Radford, J.W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al., *Learning transferable visual models from natural language supervision*, Proc. International Conference on Machine Learning (2021), 8748–8763.
23. M.M. Abdollah Pour, P. Farinneya, A. Toroghi, A. Korikov, A. Pesaranghader, T. Sajed, M. Bharadwaj, B. Mavrin, and S. Sanner, *Self-supervised Contrastive BERT Fine-tuning for Fusion-Based Reviewed-Item Retrieval*, Proc. European Conference on Information Retrieval (2023), 3–17.
24. H. Luo, L. Ji, M. Zhong, Y. Chen, W. Lei, N. Duan, and T. Li, *Clip4clip: An empirical study of clip for end-to-end video clip retrieval and captioning*. — Neurocomputing **508** (2022), 293–304.
25. V.A. Romanov and V.V. Ivanov, *Comparison of graph embeddings for source code with text models based on CNN and CodeBERT architectures*. — Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS) **35**(1) (2023), 237–264.

26. D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y.K. Li, et al., *DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence*, ArXiv preprint arXiv:2401.14196 (2024).
27. K. Ganesan, *Rouge 2.0: Updated and improved measures for evaluation of summarization tasks*, ArXiv preprint arXiv:1803.01937 (2018).
28. Y. Tian, D. Krishnan, and P. Isola, *Contrastive multiview coding.* — Computer Vision—ECCV 2020: 16th European Conference, Glasgow, UK (2020), Part XI, 776–794.
29. L. Xu, H. Xie, S.-Z.J. Qin, X. Tao, and F.L. Wang, *Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment*, ArXiv preprint arXiv:2312.12148 (2023).
30. M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, *Exploring parameter-efficient fine-tuning techniques for code generation with large language models*, ArXiv preprint arXiv:2308.10462 (2023).
31. Y. Yu, C.-H.H. Yang, J. Kolehmainen, P.G. Shivakumar, Y. Gu, S.R.R. Ren, Q. Luo, A. Gourav, I.-F. Chen, Y.-C. Liu, et al., *Low-rank adaptation of large language model rescoring for parameter-efficient speech recognition*, Proc. 2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU) (2023), 1–8.
32. Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen, and T. Zhao, *Adaptive budget allocation for parameter-efficient fine-tuning*, ArXiv preprint arXiv:2303.10512 (2023).
33. H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C.A. Raffel, *Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning.* — Advances in Neural Information Processing Systems **35** (2022), 1950–1965.
34. B. Lester, R. Al-Rfou, and N. Constant, *The power of scale for parameter-efficient prompt tuning*, ArXiv preprint arXiv:2104.08691 (2021).
35. C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M.R. Lyu, *No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence*, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2022), 382–394.
36. H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, *Rosf: Leveraging information retrieval and supervised learning for recommending code snippets.* — IEEE Transactions on Services Computing **12**, No. 1 (2016), 34–46.
37. M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, *Bimodal modelling of source code and natural language*, Proc. International Conference on Machine Learning (2015), 2123–2132.
38. Q. Chen and M. Zhou, *A neural framework for retrieval and summarization of source code*, Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering (2018), 826–831.
39. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., *Graphcodebert: Pre-training code representations with data flow*, ArXiv preprint arXiv:2009.08366 (2020).
40. A. Neelakantan, T. Xu, R. Puri, A. Radford, J.M. Han, J. Tworek, Q. Yuan, N. Tezak, J.W. Kim, C. Hallacy, et al., *Text and code embeddings by contrastive pre-training*, ArXiv preprint arXiv:2201.10005 (2022).

41. Y. Wang, W. Wang, S. Joty, and S.C.H. Hoi, *Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*, ArXiv preprint arXiv:2109.00859 (2021).
42. J. Gu, Z. Chen, and M. Monperrus, *Multimodal representation for neural code search*, Proc. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2021), 483–494.
43. S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, *Retrieval-augmented generation for code summarization via hybrid GNN*, ArXiv preprint arXiv:2006.05405 (2020).
44. J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, *A novel neural source code representation based on abstract syntax tree*, Proc. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (2019), 783–794.

Innopolis University,
Innopolis, Russia

Поступило 15 ноября 2024 г.

E-mail: {k.galliamov,l.khaertdinova,k.denisova}@innopolis.university