Better Language Models of Code through Self-Improvement

Hung Quoc To[♣,*], **Nghi D. Q. Bui**[♦,*], **Jin Guo**[♠], **Tien N. Nguyen**[◇]
[♣] FPT Software AI Center, [♦]School of Information Systems, Singapore Management University
[♠]School of Computer Science, McGill University, Canada
[◇]School of Engineering and Computer Science , The University of Texas at Dallas, USA
hungtq29@fsoft.com.vn, dqnbui.2016@smu.edu.sg, jguo@cs.mcgill.ca, tien.n.nguyen@utdallas.edu

Abstract

Pre-trained language models for code (PLMCs) have gained attention in recent research. These models are pre-trained on large-scale datasets using multi-modal objectives. However, fine-tuning them requires extensive supervision and is limited by the size of the dataset provided. We aim to improve this issue by proposing a data augmentation framework using knowledge distillation. Our framework utilizes knowledge gained during the pre-training and fine-tuning stage to generate pseudo data, which is then used as training data for the next step. We incorporate this framework into the state-of-the-art language models, such as CodeT5, CodeBERT, and UnixCoder. The results show that our framework significantly improves PLMCs' performance in sequence-generation tasks, such as code summarization and code generation in the CodeXGLUE benchmark.

1 Introduction

Pre-trained models for code (PLMCs), such as CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021), UniXCoder (Guo et al., 2022), and DISCO (Ding et al., 2022), have become the foundation to solve many practical software engineering tasks such as code summarization, code translation, program repair. Those PLMCs, like large language models (LLMs), are typically first pretrained on very large-scale datasets with a variety of multi-modal objectives under a self-supervised training style. They can then be fine-tuned using task-specific datasets in a supervised training style.

We hypothesise that, while fine-tuned models may not achieve peak performance, PLMCs can produce reasonable outputs that can be regarded as high quality data because they have been pretrained on large scale datasets, and that such data can be leveraged as additional high-quality training data. Our framework utilizes the self-improvement capability of PLMCs through an simple data augmentation step. This approach is particularly useful for tasks involving code-related sequence generation, such as code summarization and code generation. Our method involves fine-tuning a PLMC on a downstream dataset, allowing the model to gain knowledge about the task. The model then generates pseudo outputs, which are used in conjunction with the original training data to train for the next epoch. Our framework is similar to sequence-level knowledge distillation, but our approach focuses on improving model performance without compressing the model by utilizing the same technique.

Our empirical evaluation results show that our framework significantly improves the state-of-the-arts PLMCs, including CodeBERT, CodeT5, UniXCoder with significant margins. In short, we summarize our contributions as follows.

- We present a simple self-improvement framework and show how it can be easily adapted to PLMCs for the task of code-related sequence generation.
- We conduct extensive evaluation on two tasks: code summarization and code generation, and compare it with the well-known, state-of-the-art PLMCs. The results show that our framework consistently improvesover all PLMCs by a significant margin in those tasks.
- We provide analysis and explanations on how utilizing a simple framework consistently improves the performance of PLMCs.

2 Related Work

Exposure bias and hallucination in Sequence Generation Tasks The exposure bias problem is used to measure the difference between the training and inference phases for sequence generation

[*]Equal contribution. Listing order is based on the alphabetical ordering of author surnames.

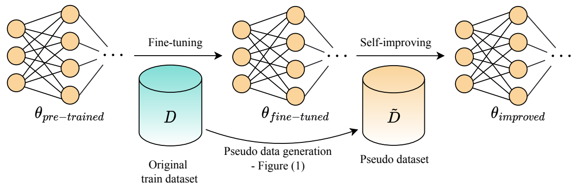

Figure 1: Overall training pipeline.

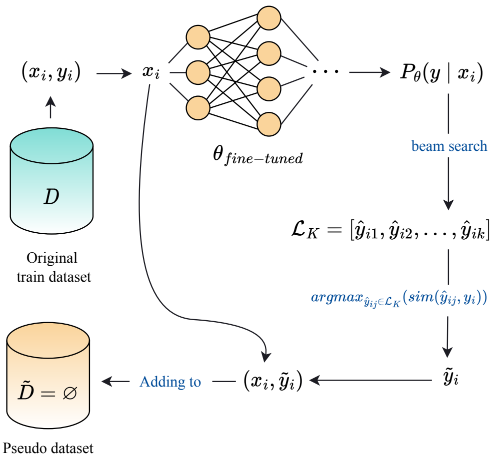

Figure 2: Demonstrating the process of generating pseudo dataset in our work.

models. Previous work has attempted to reduce exposure bias in training phase (Bengio et al., 2015; Ranzato et al., 2015; Wiseman and Rush, 2016; Wang and Sennrich, 2020). In the sense that we are attempting to close the gap between inference and training through a simple self-improvement step, the exposure bias is very close to our problem.

Code understanding and generation Code learning problems have recently emerged as one of the primary tasks for assessing the capability of language models. Most recent code models are pretrained on multi-modal objectives before being fine-tuned on specific downstream tasks (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021; Guo et al., 2022; Ding et al., 2022).

Knowledge Distillation Knowledge distillation is the process of transferring knowledge from a large unwieldy model or set of models to a single smaller model that can be practically deployed under real-world constraints, and such smaller model can usually keep the same performance or even better than the original model (Hinton et al., 2015; Kim and Rush, 2016; Wang et al., 2020; Chen et al., 2020; Mukherjee et al., 2021). We perform an additional self-improvement step to improve the original model without using external resources, our work is relevant to knowledge distillation.

3 Method

Our method utilizes three different sets of model parameters: $\theta_{pre-trained}$, $\theta_{fine-tuned}$, and $\theta_{improved}$. Each corresponds to the stage of the model parameters after pre-trained, fine-tuned, and self-improved, respectively.

Usually, models are pretrained on large scale corpora, resulting in a pre-trained checkpoint $\theta_{pre-trained}$. These pre-trained models are then fine-tuned on a specific downstream dataset D using a supervised-learning approach, resulting in a set of fine-tuned parameters $\theta_{fine-tuned}$. Our investigation revealed that model performance can be further improved if we continue to fine-tuned these parameters on an augmented version of D. As depicted in Figure 1, our proposal for self-improvement is the final step in the overall training flow. Specifically, we propose a data augmentation process and an extra fine-tuning step in addition to the pre-training and fine-tuning paradigm. The process of augmenting the dataset is illustrated in Figure 2. We also give a detailed algorithm for this process in the Appendix. For each training example (x_i, y_i), where x_i and y_i are the i^{th} source and target sequences in the train dataset D, we first use beam search to generate a list of K-best predictions L_K. This list contains k predictions, where k is the beam size.

We then evaluate the similarity of each prediction $\hat{y}_{ij}$ and its corresponding ground truth sequence y_i using a similarity function sim based on BLEU score. The best prediction with highest similarity is then selected $\tilde{y}_i = \text{argmax}_{\hat{y}_{ij} \in \mathcal{L}_K}(sim(\hat{y}_{ij}, y_i))$. In the last step, we add the pair of sequences $(x_i, \tilde{y}_i)$ into a new empty dataset $\tilde{D}$. We call this new dataset the *augmented dataset or pseudo dataset* interchangeably in the rest of the paper. The next step requires fine-tuning $\theta_{fine-tuned}$ on $\tilde{D}$ until convergence. We call this new stage of model parameters $\theta_{improved}$. Note that the index j in $\hat{y}_{ij}$ denotes the j^{th} prediction in the beam search for the i^{th} example in the dataset, not the j^{th} token of the predicted sequence. Additionally, only train dataset D is augmented, while the validation and test dataset are kept unchanged for evaluation purpose.

4 Experimental Setup

Our goal is to show that for any of the fine-tuned model for a sequence generation task (F-PLMC),

Models	Beam sizes	Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
RoBERTa	10	(Liu et al., 2019)	11.17	11.90	17.72	18.14	16.47	24.02	16.57
PLBART	10	(Ahmad et al., 2021)	14.11	15.56	18.91	19.30	18.45	23.58	18.32
PolyglotCodeBERT	10	(Ahmed and Devanbu, 2021)	14.75	15.80	18.77	18.71	20.11	26.23	19.06
CodeBERT	1	Baseline	12.04	14.73	17.58	18.47	17.62	23.44	17.31
		Self-Improved	**12.40**	**15.44**	**18.52**	**19.09**	**18.31**	**24.46**	**18.04**
	5	Baseline	12.31	15.76	18.12	19.09	18.37	24.85	18.08
		Self-Improved	**12.55**	**16.41**	**18.69**	**19.50**	**18.88**	**25.21**	**18.54**
	10	Baseline	12.22	15.78	18.01	19.09	18.42	25.06	18.10
		Self-Improved	**12.52**	**16.39**	**18.66**	**19.50**	**18.92**	**25.28**	**18.54**
CodeT5 (base)	1	Baseline	14.80	15.57	19.57	19.86	19.87	25.33	19.17
		Self-Improved	**15.46**	**16.22**	**20.12**	**20.36**	**20.25**	**26.25**	**19.78**
	5	Baseline	15.23	16.18	19.95	20.42	20.26	26.11	19.69
		Self-Improved	**15.60**	**16.51**	**20.26**	**20.59**	**20.44**	**26.46**	**19.97**
	10	Baseline	15.29	16.18	19.95	20.42	20.26	26.10	19.70
		Self-Improved	**15.70**	**16.47**	**20.28**	**20.58**	**20.45**	**26.58**	**20.00**
UniXCoder (base)	1	Baseline	14.83	15.39	18.95	18.66	19.37	24.99	18.70
		Self-Improved	**15.36**	**15.83**	**19.42**	**19.13**	**20.04**	**26.05**	**19.31**
	5	Baseline	15.17	15.93	19.52	19.25	20.18	26.45	19.42
		Self-Improved	**15.37**	**15.95**	**19.73**	**19.55**	**20.45**	**26.69**	**19.62**
	10	Baseline	14.74	15.84	19.31	19.12	20.11	26.75	19.31
		Self-Improved	**15.37**	**15.96**	**19.73**	**19.56**	**20.44**	**26.79**	**19.63**

Table 1: Results on code summarization evaluated with smoothed BLUE-4. The "Overall" column presents average scores over six programming languages. The bolded numbers represent the best scores (higher is better) when comparing between Baseline and Self-Improved for each model and each value of beam size.

after applying our self-improvement method (S-PLMC), the result improves.

Dataset and Downstream Tasks To achieve our goal of enhancing the code-related sequence generation task, we selected code summarization and code generation as our experimental areas. To evaluate these tasks, we utilized the CodeXGLUE benchmark (Lu et al., 2021), which comprises various datasets for various code understanding and code generation tasks. Specifically, we utilized the code summarization and code generation datasets from CodeXGLUE and disregarded the other ones.

Baseline Models We chose CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), and UniXCoder (Guo et al., 2022) as baseline models. Each model represents a distinct neural architecture style. CodeBERT abides to the Bidirectional Transformer architecture, which is a well-known PLMCs, despite the fact that it may not produce the best results for the tasks in CodeXGLUE. CodeT5 and UniXCoder are the two PLMCs that achieve state-of-the-arts performance on the CodeXGLUE benchmark. CodeT5 is an encoder-decoder architecture that follows the Seq2Seq learning style by following T5. UniXCoder, on the other hand, is a unified language model. It can behave as an encoder-only, decoder-only, or encoder-decoder model by modifying the masked attention matrices inside each transformer layer.

Evaluation Metric We follow CodeXGLUE benchmark in employing evaluation metrics for each task. Smoothed BLEU-4 (Lin and Och, 2004) is used as the evaluation metric for code summarization. For code generation, smoothed BLEU-4, CodeBLEU, and exact match (EM) are employed. We aim to improve all of these metrics after apply our self-improvement method into the PLMCs.

Implementation We carefully selected the checkpoints for CodeBERT [1], CodeT5 [2], and UniXCoder [3] based on the corresponding tasks. It is important to note that not all of these methods have released fine-tuned checkpoints. CodeT5 stands out as the only model to have released checkpoints for code summarization and code generation tasks. Conversely, CodeBERT and UniXCoder only offer training scripts. When checkpoints were not available, we employed the provided training scripts to fine-tune the pretrained models until we obtained results comparable to those reported in the original research. Additionally, CodeT5 and UniXCoder may have different checkpoint options with varying model sizes, such as *small*, *base*, and *large*. We selected the *base* version for both CodeT5 and UniXCoder.

[1] https://github.com/microsoft/CodeBERT/tree/master/CodeBERT

[2] https://github.com/salesforce/CodeT5

[3] https://github.com/microsoft/CodeBERT/tree/master/UniXcoder

Models	Beam sizes	Methods	EM	BLEU	CodeBLEU
CodeGPT	10	(Lu et al., 2021)	20.10	32.79	35.98
PLBART	10	(Ahmad et al., 2021)	18.75	36.69	38.52
CodeT5 (base)	1	Baseline	21.75	39.00	41.64
		Self-Improved	**22.40**	**39.75**	**42.14**
	5	Baseline	21.10	40.67	43.59
		Self-Improved	**22.40**	**41.61**	**44.06**
	10	Baseline	22.10	39.59	43.78
		Self-Improved	**22.35**	**41.85**	**44.49**
UniXCoder (base)	1	Baseline	21.50	38.28	40.85
		Self-Improved	**22.10**	**38.56**	**40.96**
	3	Baseline	22.05	37.53	40.11
		Self-Improved	**22.30**	**37.88**	**40.42**
	5	Baseline	22.00	37.18	39.78
		Self-Improved	**22.30**	**37.49**	**40.05**

Table 2: Results on code generation evaluated with EM, BLEU, and CodeBLEU. The bolded numbers represent the best scores (higher is better for all metrics) when comparing between Baseline and Self-Improved for each model and each value of beam size.

5 Hyperparameter selection

We keep all the values of hyperparameters as in the training script for each model, except that we increase the batch size in order to utilized completely memory of a NVIDIA A100 80GB.

6 Evaluation Results

The results of our code summarization task are presented in Table 1. The "Beam sizes" column indicates the beam size used in the beam search algorithm, while the "Methods" column indicates whether or not our self-improved algorithm was utilized. We also included other models as references to compare the relative improvement of our model. On average, we observed an average of 0.76 BLUE score increase in performance across all languages. This improvement was consistent across various beam sizes (1, 5, 10), which confirms the effectiveness of our self-improved approach across a wide range of PLMCs. When comparing our model to other strong baselines, we found that our method improved the performance of CodeBERT for JavaScript from 15.78 to 16.39, surpassing the performance of PolyglotCodeBERT (15.80). This highlights the benefit of our self-improved method in improving weak models. The results of our code generation study are presented in Table 2, the performance increase by 0.81 BLUE scores on average. When using EM and CodeBLEU, the improvement also increases consistently.

7 Ablation Study

In this section, we examine the factors that influence the improvement achieved by $\theta_{improved}$ as compared to $\theta_{fine-tuned}$ through code summarization. We define r_1 as the difference in performance

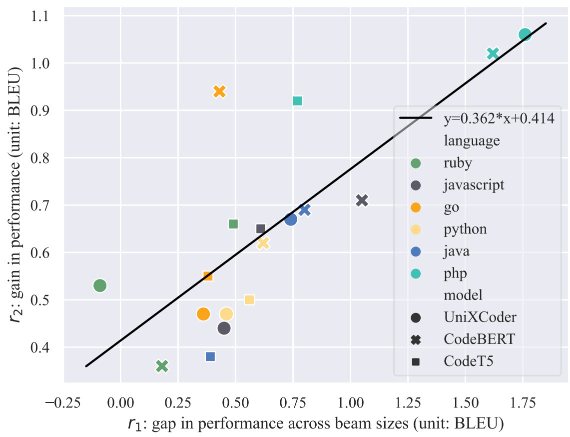

Figure 3: Scatter plot visualizing performance gap (in BLEU score) infered by different beam sizes (i.e 10 and 1) of $\theta_{fine-tuned}$ vs. performance gained (in BLEU score) by $\theta_{improved}$ infered with beam size of 1

measured by BLEU between inferencing with a beam size of 10 and inferencing with a beam size of 1. Additionally, we define r_2 as the improvement in BLEU when inferencing with the same beam size of 1 between $\theta_{fine-tuned}$ and $\theta_{improved}$. By evaluating these values across a variety of beam sizes and programming languages in the code summarization dataset, we are able to visualize the results in Figure 3. Additionally, we have calculated the Pearson Correlation score, which is 0.77, indicating a strong correlation between r_1 and r_2. Our analysis demonstrates that a larger r_1 is correlated with a better r_2, suggesting that our method is more likely to yield better overall performance when r_1 is large. We believe this insight is a crucial finding as it provides a simple indicator of the model's fully trained capability.

8 Conclusion

We introduced a self-improvement technique as a final fine-tuning step to enhance model performance. Our experiments showed that this method, when applied to popular pre-trained code models (CodeBERT, CodeT5, and UniXCoder), significantly improves performance on code summarization and code generation tasks. We also provided insights on when this method is most effective in improving PLMCs. We intend to implement our technique in larger-scale models and other tasks, and believe it is an efficient way to optimize the capabilities of any code language model without the need for extensive architecture modifications or large-scale dataset assembly. We leave all of these investigations for the future.

Limitations

We discuss a few limitations of our work. One limitation of Self-Improved is its complexity in usage. The process of generating pseudo data involves generating predictions for the entire training dataset with a large beam size, resulting in a time complexity of $O(nk)$, where n is the train dataset size and k is the beam size. Additionally, the fine-tuning step to derive $\theta_{improved}$ also adds a significant amount of computational complexity. This limitation is discussed in Section 7 to weigh the performance benefits of our method against the computational sacrifices. Another limitation is that Self-Improved has only been applied to encoder-decoder models in this work. However, it is also applicable to other types of auto-regressive models, including encoder-only models, which are commonly used for tasks such as code completion (Radford et al., 2019; Lu et al., 2021; Guo et al., 2022). Further research into these applications is left for future work.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Toufique Ahmed and Premkumar T. Devanbu. 2021. Multilingual training for software engineering. *CoRR*, abs/2112.02043.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems*, 28.

Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E. Hinton. 2020. Big self-supervised models are strong semi-supervised learners. *CoRR*, abs/2006.10029.

Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards learning (dis)-similarity of source code from program contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, Dublin, Ireland. Association for Computational Linguistics.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*.

Yoon Kim and Alexander M. Rush. 2016. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1317–1327, Austin, Texas. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Subhabrata Mukherjee, Ahmed Hassan Awadallah, and Jianfeng Gao. 2021. Xtremedistiltransformers: Task transfer for task-agnostic distillation. *CoRR*, abs/2106.04563.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*.

Chaojun Wang and Rico Sennrich. 2020. On exposure bias, hallucination and domain shift in neural machine translation. *arXiv preprint arXiv:2005.03642*.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *CoRR*, abs/2002.10957.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Sam Wiseman and Alexander M Rush. 2016. Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960*.

A Algorithms

Algorithm 1 Pseudo Data Generation

Input:

- $\theta_{fine-tuned}$, the fine-tuned model checkpoint on a specific task $T \in$ {code summarization, code generation, code translation, .etc. }.
- $D = \{(x_i, y_i) \mid i = \overline{1, n}\}$, the train dataset on which the $\theta_{fine-tuned}$ is fine-tuned.
- B_k denotes the $beamsearch$ algorithm with beam size of k. It returns a list of k best sequences as prediction.

Output:

- Pseudo dataset $\tilde{D}$

```
procedure GeneratingPseudoData
    D̃ ← ∅
    for each datapoint (x_i, y_i) ∈ D do:
        L_K ← B_k(P_θfine-tuned(y | x_i))
        In other words, L_K = [ŷ_i1, ŷ_i2, ..., ŷ_ik]
        ỹ_i ← argmax_{ŷ_ij ∈ L_K}(sim(ŷ_ij, y_i))
        Adding (x_i, ỹ_i) → D̃
    end for
    return D̃
end procedure
```

$$\mathcal{L}_K \leftarrow B_k(P_{\theta_{fine-tuned}}(y \mid x_i))$$

$$\mathcal{L}_K = [\hat{y}_{i1}, \hat{y}_{i2}, ..., \hat{y}_{ik}]$$

$$\tilde{y}_i \leftarrow \underset{\hat{y}_{ij} \in \mathcal{L}_K}{\text{argmax}}(sim(\hat{y}_{ij}, y_i))$$