Improving Repository-level Code Search with Text Conversion

Mizuki Kondo[1] **Daisuke Kawahara**[1] **Toshiyuki Kurabayashi**[2]
[1]Waseda University [2]NTT Laboratory
{kondmiznotfound@toki., dkw@}waseda.jp toshiyuk.kurabayashi@ntt.com

Abstract

The ability to generate code using large language models (LLMs) has been increasing year by year. However, studies on code generation at the repository level are not very active. In repository-level code generation, it is necessary to refer to related code snippets among multiple files. By taking the similarity between code snippets, related files are searched and input into an LLM, and generation is performed. This paper proposes a method to search for related files (code search) by taking similarities not between code snippets but between the texts converted from the code snippets by the LLM. We confirmed that converting to text improves the accuracy of code search.

1 Introduction

Currently, the code generation capability of large language models (LLMs) has significantly improved. The accuracy of understanding and generating individual pieces of code has become high. However, there is little research at the repository level, which is closer to actual software development, and the ability of LLMs to generate code at the repository level is very low. LLMs' best debugging accuracy at the repository level is only 1.96% on the debugging benchmark SWE-bench (Jimenez et al., 2023).

Code-related tasks at the repository level require referring to many files. However, most LLMs are based on Transformer (Vaswani et al., 2017), which has a limitation on input length, preventing the input of many files. Therefore, methods have been proposed to search for relevant code snippets based on similarity and input only these into LLMs (Liu et al., 2023). The accuracy of code search is low on SWE-bench and RepoBench (Liu et al., 2023), repository-level code completion and search benchmark.

This paper focuses on improving the code search method for code completion tasks. The code completion task is a task that predicts the next line of unfinished code. For predicting the next line, multiple related files are provided based on the dependencies between files obtained by parsing the unfinished code. In this paper, the unfinished code is referred to as the **target code**, and the multiple related files are referred to as **code candidates**. The task of selecting the relevant code from the code candidates based on the information of the target code is referred to as code search.

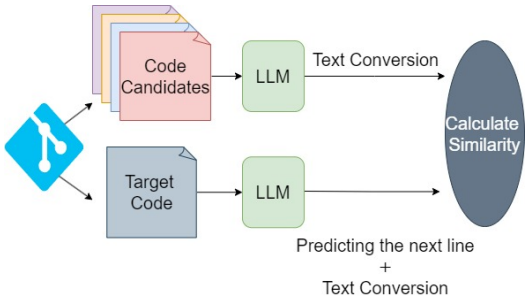

Figure 1: Overview of our proposed method.

Existing studies obtain features, such as embeddings from language models, of both the target code and code candidates to calculate similarity for code search (Liu et al., 2023). RepoCoder (Zhang et al., 2023) searches for code using such methods, generates code once, and then re-searches and generates the output code using the generated code. In this study, instead of directly taking the similarity between code snippets, similarity is calculated after transforming the code with an LLM. Code candidates are converted to text, and the target code is converted to text or to the prediction of the next line and its explanation, which is a combination of RepoCoder's method and text conversion. Figure 1 shows the flow of text conversion.

We confirmed an improvement in the accuracy of code search in code completion experiments using our proposed method. We also examined the prompts used for text conversion with LLMs.

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 130–137

2 Related work

2.1 LLMs trained on code

In recent years, there has been an increase in Transformer-based LLMs trained on code, including CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022) for encoder models, Codex (Chen et al., 2021), StarCoder (Li et al., 2023), and Code Llama (Roziere et al., 2023) for decoder models, CodeT5+ (Wang et al., 2023) for encoder-decoder models. Especially, the development of decoder models has been remarkable, and their code generation capabilities have significantly improved. These models exhibit high accuracy in generating individual pieces of code and perform well on code benchmarks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021).

2.2 Repository-level studies

In software development, multiple files are used rather than a single file. To deal with real-world software development tasks, studies have been conducted on repositories such as GitHub (Just et al., 2014).

With the improvement of code generation capabilities of LLMs, there has been an increase in studies and benchmarks at the repository level based on LLMs (Jimenez et al., 2023; Liu et al., 2023; Zhang et al., 2023; Ding et al., 2022; Shrivastava et al., 2023). RepoCoder (Zhang et al., 2023) improved accuracy by repeating search and generation twice in code completion tasks. RepoBench (Liu et al., 2023) is a benchmark for code completion, consisting of three tasks: code search, code completion, and two pipeline tasks. SWE-bench (Jimenez et al., 2023) is a benchmark that collected GitHub issues and corresponding pull requests from Python repositories to compete on how well LLMs can solve real-world problems.

The accuracy on SWE-bench is significantly low. Compared to studies on single code, those at the repository level are less conducted in terms of the number of methods and datasets.

3 Proposed method

3.1 Overview

In previous methods of code search, the target code and code candidates are input directly into a language model.[1] Then, the similarity between the target code and a code candidate is calculated based on their embeddings (Liu et al., 2023). In our proposed method, code is converted into text by an LLM, and its embeddings are obtained using a language model. For calculating similarity, we try two methods: the cosine similarity of mean embeddings and BERTScore (Zhang et al., 2020). An example of text conversion of code candidates is shown in Figure 2.

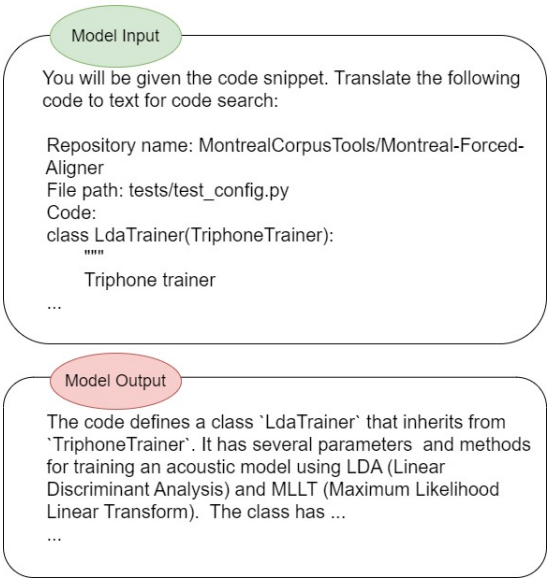

Figure 2: Code candidate conversion.

Furthermore, in addition to converting the target code into text using an LLM, we also propose a method that combines our proposed method with the method of RepoCoder (Zhang et al., 2023). In RepoCoder, the LLM is used to predict the next line once, and the predicted line is used for re-searching. In contrast, we propose adding explanations to the prediction of the next line. An example of the conversion that outputs an explanation in addition to the prediction of the target code is shown in Figure 3.

3.2 Prompt design

It is known that prompts have a significant impact on the output of LLMs (Wei et al., 2022). Therefore, we create and test several prompts. In addition to manually created prompts, we propose automatic prompts that are generated by the LLM itself. An example of an automatic prompt is shown in Figure 4. The example in Figure 4 instructs the LLM

[1] In this paper, models to be converted to embeddings are called "language models" rather than LLMs because of their small model size.

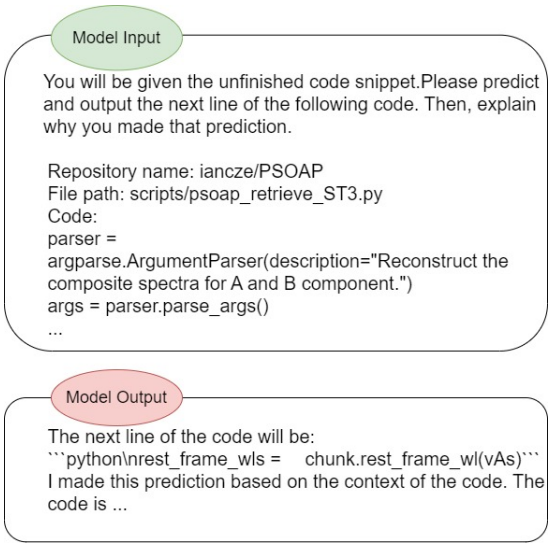

Figure 3: Target code conversion. In this case, an LLM converts the target code to a prediction of the next line and an explanation of the prediction.

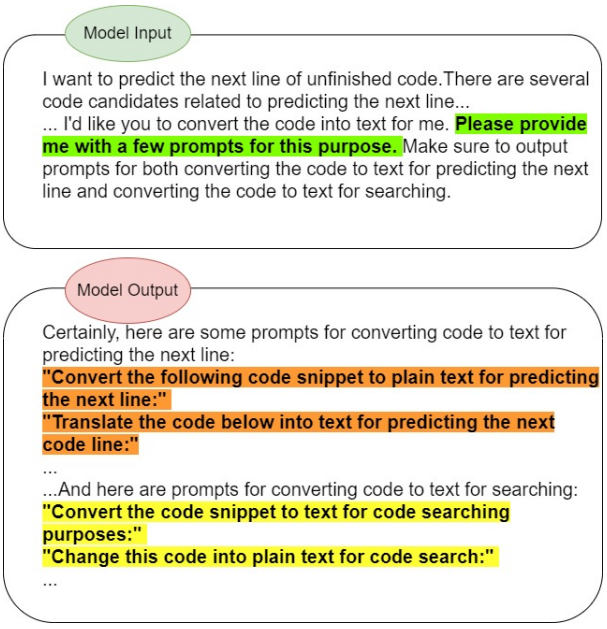

Figure 4: Auto Prompt. The green-highlighted section instructs an LLM to generate a prompt, the orange-highlighted section is an output of a text conversion prompt for the target code, and the yellow-highlighted section is an output of a prompt for text conversion for the code candidates.

to describe a certain situation and then output a prompt. This results in the generation of multiple prompt candidates.

4 Experiments

4.1 Experimental settings

Code candidates and target code expressions We examine several patterns of text conversion by LLMs in the code completion tasks. The code candidates are evaluated in three types: the original code without conversion, text conversion using prompts created by humans, and text conversion using automatic prompts. In addition to these three types, the target code is evaluated in a total of five types, including the method of RepoCoder (Zhang et al., 2023) described in Section 3.1 and the proposed method. The manually created prompts and the automatic prompts were determined by trying several patterns and adopting the one with the best accuracy on a small dataset. The actual prompts used are shown in Table 1.

Used models / datasets Gpt-3.5-turbo[2] is used as the LLM for text conversion. The temperature parameter is set to 0 to ensure consistent generation. The models used for conversion to embeddings are RoBERTa[3], UniXcoder[4], CodeBERT[5], and text-embedding-ada-002[6] (hereafter referred to as ada-002). For calculating the similarity between converted texts, the cosine similarity of mean embeddings and BERTScore (Zhang et al., 2020) were compared.

For evaluation, we use the Java and Python datasets of the repobench-r[7] code search task from RepoBench (Liu et al., 2023). Our evaluation is conducted with a set of 8,000 pieces of target code and code candidates under the settings of "XFF" and "Easy". "XFF" is the setting where the next line to be predicted in the target code is the first one to refer to external code. "Easy" is the task where, on average, there are 6.6 files for Java and 6.7 files for Python among the code candidates. The code candidates are provided based on the dependencies between files obtained by parsing the unfinished code.

[2] https://platform.openai.com/docs/models/gpt-3-5

[3] https://huggingface.co/roberta-base

[4] https://huggingface.co/microsoft/unixcoder-base

[5] https://huggingface.co/microsoft/codebert-base

[6] https://platform.openai.com/docs/models/embeddings

[7] https://huggingface.co/datasets/tianyang/repobench-r

Type	Prompt
Common	You will be given the code snippet.
Human	Your task is to summarize the code into text for code retrieval. The length should be around 500 characters.
Auto	Translate the following code to text for code search:
Meta information	Repository name: Actual repository name File path: Actual file path

(a) Prompt used to convert the code candidates.

Type	Prompt
Common	You will be given the unfinished code snippet.
Human	Your task is to summarize the code into text for predicting the next line of the code. The length should be around 500 characters.
Auto	Convert the given incomplete code snippet into natural language text:
Pred	Predict the next line of the following code and output it. Make sure to only output the prediction.
Pred+Explain	Please predict and output the next line of the following code. Then, explain why you made that prediction.
Meta Information	Repository name: Actual repository name File path: Actual file path

(b) Prompt used to convert the target code.

Table 1: The prompts used for the code candidates and the target code. Type indicates the type of prompt: Common is the first prompt entered at the beginning of every prompt; Human is a human-created prompt; Auto is a prompt created by automatic prompting; Pred is a prediction of the next line; and Pred+Explain is a prediction of the next line with its explanation. Meta Information is the information entered at the end of every prompt. The prompts consist of the Common statement first, followed by the Human, Auto, Pred, or Pred+Explain statement, and finally the Meta Information. After these prompts, the code is entered.

Evaluation metrics The evaluation of the code search task follows the evaluation method of RepoBench. The metric is the percentage of correct answers that are the code candidates with the highest similarity (acc@1) and the percentage of correct answers that are included in the three most similar ones (acc@3).

4.2 Results

The evaluation results for Python are shown in Table 2, and the evaluation results for Java are presented in Appendix A. This paper discusses Table 2. Table 2 lists, from left to right, the method of calculating similarity, the model used to obtain embeddings for calculating similarity, the prompt for code candidates, and the prompt for a target code. The prompts are represented as follows: "Human" for manually created prompts, "Auto" for automatically generated prompts, "Original" for the original code, "Pred" for the prediction of the next line, and "Pred+Explain" for the prediction of the next line with an added explanation.

The proposed method is more accurate than the baseline. The highest accuracy was achieved with ada-002 for both acc@1 and acc@3, using the cosine similarity of embeddings, when the code candidate was Original and the target code was Pred+Explain. Among the publicly available models, it was achieved by UnixCoder with BERTScore, when the code candidate was Human and the target code was Pred+Explain. Compared to the baseline, where the code candidate and the target code were both Original, the accuracy of the proposed method was significantly higher, confirming its effectiveness.

Pred+Explain is highly accurate. Overall, the accuracy is good when the target code is Pred+Explain, and in most cases, it is higher than the other conversion methods. In particular, Pred, which predicts the next line, is the existing method proposed by RepoCoder, and the fact that accuracy improves by adding explanations confirms the effectiveness of the proposed method.

Retrieval	Model	Candidate	Unfinished Code									
			Human		Auto		Original		Pred		Pred+Explain	
			acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@1	acc@3
BERTScore	RoBERTa	Human	17.06	48.77	15.92	48.95	18.29	51.09	20.85	53.64	24.27	55.39
	CodeBERT	Human	16.36	48.65	15.36	48.25	18.09	50.39	19.97	52.27	21.60	53.71
	UniXcoder	Human	20.52	52.92	18.94	51.84	25.90	58.74	30.57	61.91	**31.34**	**63.09**
	RoBERTa	Auto	16.55	47.75	15.69	47.72	17.44	49.44	18.09	50.45	20.49	50.99
	CodeBERT	Auto	16.54	47.72	15.84	46.85	17.60	49.71	17.64	50.21	18.24	49.90
	UniXcoder	Auto	19.79	52.32	17.47	50.57	24.94	58.80	28.79	60.76	29.74	61.52
	RoBERTa	Original	16.59	47.92	15.91	47.36	17.24	47.91	17.97	49.57	19.12	50.29
	CodeBERT	Original	16.21	46.99	15.74	46.50	16.84	48.51	17.86	49.34	18.26	49.06
	UniXcoder	Original	19.86	52.34	18.02	50.81	25.00	59.06	28.79	60.72	29.37	62.12
Embedding	RoBERTa	Human	16.52	48.14	16.17	48.57	16.45	47.50	16.95	47.97	19.41	50.19
	CodeBERT	Human	15.84	47.85	16.07	47.70	15.39	47.32	16.62	47.47	18.34	48.85
	UniXcoder	Human	20.29	53.64	19.00	51.75	25.06	58.90	29.79	61.32	30.65	62.46
	ada-002	Human	19.40	52.81	17.81	51.35	28.40	61.92	33.36	64.80	33.71	65.56
	RoBERTa	Auto	16.35	47.51	15.72	47.14	15.85	47.74	16.55	47.77	17.37	47.71
	CodeBERT	Auto	16.31	47.71	15.54	47.97	15.86	46.76	16.04	47.15	16.51	46.75
	UniXcoder	Auto	19.70	52.24	17.16	50.15	24.16	58.12	27.82	60.61	29.40	61.35
	ada-002	Auto	19.14	52.42	18.32	50.07	29.06	62.49	34.55	65.38	34.00	65.58
	RoBERTa	Original	16.86	48.11	16.34	47.64	15.97	47.05	16.75	48.42	17.07	48.65
	CodeBERT	Original	16.15	46.91	16.04	46.85	15.77	46.26	15.95	47.37	16.00	47.12
	UniXcoder	Original	19.85	51.74	17.66	49.96	24.15	58.85	27.66	60.00	28.10	59.95
	ada-002	Original	19.64	52.56	17.62	50.49	27.95	63.31	33.66	65.76	**34.82**	**66.61**

Table 2: Result of the Python dataset. This table lists, from left to right, the method of calculating similarity, the model used to obtain embeddings for calculating similarity, the prompt for code candidates, and the prompt for a target code. The prompts are represented as follows: "Human" for manually created prompts, "Auto" for automatically generated prompts, "Original" for the original code, "Pred" for the prediction of the next line, and "Pred+Explain" for the prediction of the next line with an added explanation.

When the target code is Human or Auto, the accuracy is low. On the contrary, when the target code was Human or Auto, the accuracy was lower than the baseline. This is thought to be because the conversion of the code into text was done for the entire code, which reduced the information about the next line. When the target code was Original, the last three lines were input into the model, following RepoBench. This treatment is believed to have retained more information about the following line.

High accuracy was achieved for code candidate conversion through manual prompts. When we focus on the text conversion of code candidates, the overall trend in accuracy shows that Original and Auto are roughly the same, with Human having higher accuracy. This indicates that while the effectiveness of text conversion was confirmed, that of automatic prompts was not observed. The design of prompts, under the conditions of this experiment, resulted in higher accuracy when done manually, making the automatic creation of prompts a challenge for future work.

UniXcoder and ada-002 are highly accurate. When evaluating the accuracy for each model, UniXcoder and ada-002 had high overall accuracy, while RoBERTa and CodeBERT had low accuracy for all prompts. The trends for CodeBERT and UniXcoder were similar to those reported in RepoBench, with CodeBERT having low accuracy and UniXcoder having high accuracy. RoBERTa, which is not pre-trained on code, was supposed to have higher accuracy because the code is converted into text through text conversion. However, the result was low. Ada-002 had high accuracy in the CodeSearchNet (Husain et al., 2019) dataset[8] and also achieved high accuracy in repobench-r.

BERTScore is more accurate than Embedding. For the calculation methods for similarity, BERTScore is more accurate than Embedding when comparing the same models. This is because BERTScore retrieves similarity for each token when calculating similarity, resulting in wide-coverage information. However, the highest accuracy was achieved with Embedding's ada-002, both for acc@1 and acc@3, when the code candidate was Original and the target code was Pred+Explain. It should be noted that BERTScore cannot be applied to proprietary models such as ada-002.

Analysis of the computational resources required for text conversion Text conversion of code candidates requires all files to be converted to text by an LLM. However, by creating and caching embeddings of the converted text, only a one-time conversion is required, which requires relatively few computational resources.

The text conversion of the target code generates a prediction of the next line and its explanation to

[8] https://openai.com/blog/new-and-improved-embedding-model

perform a code search. This requires more computational resources than simply predicting the next line. However, the explanations are often short, and thus the computational resources are not used excessively.

5 Conclusion

In this study, we proposed a method for code search in code completion tasks, which involves converting code into text to obtain similarity. Additionally, we proposed an automatic prompting method that creates prompts for LLMs. While an improvement in accuracy was confirmed for text conversion, no improvement in accuracy was observed for automatic prompting.

We hope that this study will contribute to the development of code generation tasks at the repository level. In the future, we aim to apply text conversion not only to tasks other than code completion tasks, such as debugging, but also beyond repository-level tasks.

Acknowledgements

This work was conducted in collaboration with NTT Laboratory and Waseda University.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

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.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

A Result of the Java dataset

Retrieval	Model	Candidate	Unfinished Code									
			Human		Auto		Original		Pred		Pred+Explain	
			acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@1	acc@3	acc@1	acc@3
BERTScore	RoBERTa	Human	13.90	45.26	12.96	43.82	16.86	49.22	20.21	52.31	20.75	52.97
	CodeBERT		14.55	46.27	13.32	45.10	17.11	48.99	18.84	51.22	18.61	50.70
	UniXcoder		13.87	45.79	13.19	43.89	20.70	53.96	24.95	56.76	23.55	56.89
	RoBERTa	Auto	15.11	46.64	14.19	45.47	16.75	48.61	19.59	51.10	19.76	52.22
	CodeBERT		15.42	47.95	14.34	45.75	16.64	48.46	18.16	49.87	18.25	49.76
	UniXcoder		14.12	45.47	12.97	44.09	19.80	53.69	23.40	55.32	22.91	55.72
	RoBERTa	Original	14.75	46.65	13.99	45.74	16.46	49.45	17.31	50.85	16.96	50.30
	CodeBERT		14.92	47.29	14.25	46.85	16.61	49.72	17.20	50.45	17.22	49.34
	UniXcoder		14.85	46.27	13.55	45.14	20.24	54.04	24.06	57.62	22.75	56.30
Embedding	RoBERTa	Human	14.80	46.30	14.57	45.54	16.39	49.60	16.96	49.82	18.27	50.26
	CodeBERT		14.95	47.46	14.89	46.19	15.85	48.29	16.67	49.21	17.04	49.14
	UniXcoder		14.00	46.00	13.10	44.11	20.20	53.91	24.21	56.62	23.46	56.69
	ada-002		12.90	44.74	12.34	43.15	21.61	55.97	25.81	58.71	24.70	58.86
	RoBERTa	Auto	14.99	46.56	15.17	45.92	16.45	48.86	17.19	48.99	17.37	49.66
	CodeBERT		15.62	47.46	14.81	46.29	15.87	48.60	16.72	48.35	16.70	48.57
	UniXcoder		14.14	45.95	13.15	44.16	20.32	54.30	23.04	55.67	22.96	56.21
	ada-002		13.39	44.99	12.46	43.71	22.25	56.95	26.44	59.21	25.90	59.16
	RoBERTa	Original	14.77	46.34	14.21	46.26	16.21	47.79	15.96	47.80	15.30	47.06
	CodeBERT		15.42	47.16	14.99	47.10	15.47	47.86	16.07	48.14	15.54	47.62
	UniXcoder		14.89	47.16	13.84	45.94	20.05	52.97	22.66	55.92	21.00	54.69
	ada-002		13.12	45.12	12.16	43.56	22.10	57.20	27.91	61.24	26.41	60.52

Table 3: Result of the Java dataset. The trend is generally the same as Python, but in many cases, Pred is more accurate than Pred+Explain. The proposed method is effective because the results are better when the code candidates are converted to text.