Code snippets insertion approach for code completion task

Anonymous ACL submission

Abstract

The purpose of the code completion task is to continue the following line of the source code. This feature helps developers to write code faster and with fewer errors by providing suggestions for completing the current line of code based on the context and the available libraries and functions. One common but resource-expensive approach to improve model performance on the code completion task is to fine-tune Large Language Models (LLMs). This paper introduces a technique that involves feeding the additional input with ranked portions of code to the LLMs without additional fine-tuning. The proposed approach aims to replicate the development process of programmers by scanning project files for the required code snippets, making the process intuitive and efficient. The paper also discusses the lack of metrics for the task and puts forward a novel metric ClickScore, as well as a new code benchmark RealCode for the code completion task. The paper compares the insertion technique of code snippets with the existing state-of-the-art methods using standard and proposed metrics and demonstrates the approach's effectiveness.

1 Introduction

Code completion is widely utilized (Svyatkovskiy et al., 2021) in the field of Programming Language Processing (PLP) to aid in accurately writing and extending code. It offers significant advantages to programmers by minimizing the time required to create fully functional code and systems.

In the paper, we investigate the code completion task and propose the approach to improve the quality of LLMs without necessitating additional model fine-tuning. We gathered real code data from projects on GitHub and established a benchmark to assess the quality of experiments conducted within the context of the code completion task.

The paper presents an approach that enhances the quality of LLMs with reduced time and memory overhead compared to existing approaches, particularly those involving fine-tuning models. The primary objective of the approach is to incorporate an additional small code segment, which is initially identified by ranking all the files within the project from which the source code requiring code continuation originates. Subsequently, the source code, along with the top-ranked code snippet, is inserted into the LLMs. The evaluation of the approach's quality is conducted by comparing the method that incorporates the insertion of code fragments with the traditional method without inserting fragments.

Thus, the contribution of work can be summarized as follows:

- We present an approach to code completion that does not require LLMs fine-tuning;
- We introduce and release [1] in the open-source RealCode benchmark that contains Python programming language;
- We propose the efficient ClickScore metric, which offers a more comprehensive evaluation of the effectiveness of LLMs predicting the following line.

2 Related Work

In previous studies, (Li et al., 2023a) proposed a method for training a model to link variables in code from one file to another. This approach hides irrelevant local variables and prevents distractions when substituting code snippets for input. However, a pre-ranked snippet list and time for model training are required to detect and eliminate unnecessary variables between files.

The code retrieving task is solved using ReACC: A Retrieval-Augmented Code Completion Framework (Lu et al., 2022), where the training process

[1] *The link was removed to preserve anonymity for the review period.*

of the retriever is in proposed framework ReACC. Still, such an approach requires additional fine-tuning for successful implementation, which is resource-consuming.

3 Approach

3.1 Overview

The method aims to assist LLMs in displaying more useful code and expanding their knowledge base. We propose an intuitive approach that does not require fine-tuning the LLMs. Main goal is to simulate developers' daily work by identifying meaningful code within the appropriate project files for LLMs to use. To achieve this, we **1)** pre-process the entire code space of project files 3.2, **2)** rank all file snippets and select the most relevant ones based on their similarity to the source file 3.3. Below, we will discuss the definitions of the terms used in the approach and describe the experiments conducted within this approach.

Throughout all experiments, the algorithm stays consistent, with only the method of ranking and selection snippets being subject to change. The pre-processing of snippets remain consistent across all experiments.

3.2 Pre-processing of the code data

The source code requiring the continuation of the next line is extracted from a specific project. The first stage of the approach incorporates pre-processing of the code files within the each project files space of each source code. Two components are needed to define: the query and context. The left context serves as a query (`left context`) for which the corresponding relevant context (code `snippet`) needs to be found. The code snippet has two parts: 1) `snippet` 2) `add to snip`.

Left context is the source code file taken by the last ten lines of code (see the code example 1).

```
package io.fabric8.crd.example.joke;
import io.fabric8.kubernetes.
    PrinterColumn;
public class JokeRequestSpec {
  public enum Category {
    Any,
    Misc,
    Programming,
    Dark,
    Pun,
    Spooky,
    Christmas
  }
  public enum ExcludedTopic {
    nsfw,
    religious,
    political,
```

Listing 1: Inital code (source) example.

Snippet code is a snipped piece of code that contains ten lines (i.e., ten line breaks). We do not enforce reproducibility restrictions on the trimmed code snippet. These snippets are obtained by slicing a complete code structure into small pieces with a window of ten lines. This process divides the source file into multiple `snippet codes`.

```
package io.fabric8.crd.example.joke;
import io.fabric8.kubernetes.
    PrinterColumn;
public class JokeRequestSpec {
  public enum Category {
    Any,
    Misc,
    Programming,
    Dark,
    Pun,
    Spooky,
    Christmas
```

Listing 2: Snippet code example (a small part of the code file).

Add to snip is the next five lines of code after the `snippet code`, which do not participate in ranking when choosing the best relevant snippet to submit. Add to snip is added after the snippet itself is inserted.

```
  }
  public enum ExcludedTopic {
    nsfw,
    religious,
    political,
```

Listing 3: The next five lines after the snippet code.

3.3 Ranking and snippet selection

This chapter outlines two experiments, they are employing a different method for ranking code snippets. The crucial distinction across these experiments lies in how the code snippets are ranked and selected.

In the first experiment, detailed in the 4.2 section, snippet ranking is accomplished using the BM25 algorithm. Meanwhile, in the 4.1, the code encoder generates embeddings for each code snippet, which are then indexed using the Faiss index. The final ranking is determined by measuring the dot product between the embeddings of the code snippets and left context.

After ranking and selecting the best snippet, it has been inserted into LLMs. The proposed fitting mode into LLMs is outlined below. The fitting mode is consistent across all experiments.

```

# {filename_of_best_snippet}
{best_snippet}

# {filename_of_left_context}
{left_context}
```

Listing 4: Input format for LLMs.

The best snippet is the concatenation of **snippet code** and **add to snip**.

4 Experiments

4.1 Embeddings similarity ranking

All the snippets were created based from the files that constitute the entirety of the project, from which the left context was derived in a manner consistent with the provided in 3.2.

Snippets were let through encoder CodeSage-base without fine-tuning [2] (Zhang et al., 2024) to get embeddings of size 1024 from each of them. The left context is also run through CodeSage-base. After obtaining embeddings for the left context and snippets, the Faiss index [3] are raised with all the embeddings in it.

The similarity of two embeddings (left context and snippet code) is determined using the dot product within the Faiss index. A higher similarity distance number indicates a greater similarity between the two vectors.

During validation, snippets with the same file name as the original query file should be ignored. In such cases, the distance will be highest in terms of proximity, indicating a potential leak, as the relevant snippet (context) will be the same file that is considered as the query. The final step is to select the best snippets based on similarity with dot product and feed them to the LLMs input in the format presented in Listing 4.

4.2 BM25 ranking

ElasticSearch (Elastic) [4] is a tool for bringing up the index and searching within it. Elastic is powered by the BM25 (Chen and Wiseman, 2023) algorithm. BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document, regardless of their proximity within the document. In this way the left context represents the query, while the documents consist of sliced snippet codes.

In this experiment, Elastic performed a search for relevant snippets. The algorithm was as follows: 1) pre-process the entire code space of project files 3.2 2) building an index using Elastic; 3) ranking and selecting relevant snippets with top-K based on BM25 algorithm running by index on the previous step.

The final step is to select the best snippets and feed them to the LLMs input in the format presented in 4.

5 Evaluation

5.1 RealCode benchmark

RealCode is a benchmark to perform an execution-based evaluation of LLMs code generation for real GitHub repositories containing. The dataset contains 219 Python functions [5] from 22 GitHub repositories [6] published between June and August 2023. All these functions are covered with tests in their respective repositories.

The benchmark task is to write the body of a function that is declared in a file within one of the repositories. The benchmark supplies the model with the remainder of the file or the entire repository. The metrics used for the benchmark is the Pass@k metric (Chen et al., 2021)) is employed for evaluation. We define a successful generation as one that passes the same number of tests as the actual body of a function.

5.2 ClickScore metric

The Exact Match (EM) (Wang et al., 2024) metric is commonly used in code completion to measure the percentage of correctly generated strings according to the model's predictions, including spaces and special characters. However, relying solely on this metric may not provide a comprehensive assessment of the model's quality, as it is too strict and underestimates the model's performance. Thus, we propose a new metric called the **ClickScore** (CS).

This metric reflects the number of human's keystrokes saved when a hint of possible code continuation is given. For instance, if there is a code line in Python: $a, b = 0, 1$, and the predicted line is $a, b = 0, 4$, then the EM is 0, but the CS is 0.9.

[2] https://huggingface.co/codesage/codesage-base
[3] https://github.com/facebookresearch/faiss
[4] https://www.elastic.co/elasticsearch

[5] In the paper, the term "function" includes methods in classes
[6] About 60 percent of the repositories used are related to the field of AI, LLMs, and ML

Thus, the CS shows how closely and numerically the predicted string matches the ground truth string. If EM equals 1, then CS will also equal 1. The CS metric presents a more realistic view of model prediction performance than EM.

5.3 LLMs inference

We conducted measurements on the open-source models using the aforementioned experiments to assess their performance based on CS and EM metrics on RealCode benchmark in fill in the middle (FIM) mode (Bavarian et al., 2022) with 4096 context size. Top-1 snippet codes were collected across all experiments.

Used open-source LLMs: DeepseekCoder-6.7B base model (Guo et al., 2024) [7] , CodeGemma-2B [8] and CodeGemma-7B (Team, 2024) [9] , CodeQwen-7B (Bai et al., 2023) [10].

All of them had been used without fine-tuning for the task.

Model	CS	EM
DeepSeek Coder$_{6.7B}$	0.71	0.51
DeepSeek Coder$_{6.7B\,snippet}$	**0.84**	**0.74**
CodeGemma$_{2B}$	0.61	0.15
CodeGemma$_{2B\,snippet}$	**0.75**	**0.25**
CodeGemma$_{7B}$	0.69	0.20
CodeGemma$_{7B\,snippet}$	**0.83**	**0.31**
CodeQwen$_{7B}$	0.73	0.52
CodeQwen$_{7B\,snippet}$	**0.83**	**0.74**

Table 1: Embeddings similarity ranking experiment on Code Completion by RealCode benchmark. Best model scores are bold.

Model	CS	EM
DeepSeek Coder$_{6.7B}$	0.71	0.51
DeepSeek Coder$_{6.7B\,snippet}$	**0.86**	**0.76**
CodeGemma$_{2B}$	0.61	0.15
CodeGemma$_{2B\,snippet}$	**0.78**	**0.26**
CodeGemma$_{7B}$	0.69	0.20
CodeGemma$_{7B\,snippet}$	**0.85**	**0.33**
CodeQwen$_{7B}$	0.73	0.52
CodeQwen$_{7B\,snippet}$	**0.86**	**0.76**

Table 2: BM25 experiment on Code Completion by RealCode benchmark. Best model scores are bold.

6 Results

The proposed approach is intuitive and seeks to replicate the real-life scenarios developers encounter. In practice, developers frequently reference project files directly during their coding process or utilize pre-existing code written by others and subsequently modify it. The integration of optimal snippet codes into LLMs facilitates the model's ability to assimilate the context of the file.

The analysis of ranked snippets revealed that most of them originated from files that closely resembled the left context, necessitating continuation. We do not consider it as a leakage as in practice, the developers do absolutely the same.

If we compare the results from the Embeddings similarity ranking (see Table 1) and the BM25 experiment (see Table 2), the metrics are pretty similar. The first experiment is designed to capture semantic knowledge from code using embeddings, while the second experiment aims to identify syntactically similar files. The overlap in relevant snippets between the two experiments was over 60%, indicating significant commonality.

We have also conducted an additional experiment regarding the selection between whole file and short snippet code staging. The results are presented in Table 3. The metrics indicate that employing LLMs on full files rather than short code snippets does not yield any improvement over using snippet codes.

We also conducted an evaluation of various models using the specified approach across nine programming languages. The results are documented in 1. This method consistently demonstrates an enhancement across all languages.

7 Conclusion

The success of padding the snippet into the model can be attributed to the combination of copy-paste and semantics between left context and best snippet codes. Developers often utilize existing code and modify it when writing new code. We attempted to replicate this in practice by enhancing visibility through the inclusion of the most relevant similar code snippets alongside the code provided by LLMs.

During the research, we conducted a comparison of the performance of open-source LLMs using both the fine-tuning and snippet code approaches. The results indicated that fine-tuning only resulted in a marginal 1-2 percent improvement compared to the approach without fine-tuning. This performance underscoring the substantial effectiveness of code snippets insertion approach.

[7] https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base
[8] https://huggingface.co/google/codegemma-2b
[9] https://huggingface.co/google/codegemma-7b
[10] https://huggingface.co/Qwen/CodeQwen1.5-7B

Limitations

In current approach, the substantial increase in metrics can be attributed to a significant number of copy-paste instances, which may present challenges in the context of code completion task. It is anticipated that there will be minimal requests for hints in code where obvious copies are located in other files, as the necessity for hints for developers is obviated in such cases.

Effectively managing the substantial volume of code snippets without compromising performance poses a significant challenge. One potential solution for embedding similarity experiment is outlined here A.0.1

The code in the RealCode repositories was not observed during the fine-tuning of StarCoder (Li et al., 2023b) [11] or CodeLlama (Roziere et al., 2023) [12], as these models were trained before the summer of 2023. DeepSeek Coder may have encountered this code during fine-tuning.

Training an embedder that will be tuned for a specific code completion task is one way to improve approach.

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, Joshua 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. *CoRR*, abs/2107.03374.

Xiaoyin Chen and Sam Wiseman. 2023. Bm25 query augmentation learned end-to-end. *ArXiv*, abs/2305.14087.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023a. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2124–2135. IEEE.

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

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*.

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

Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE.

CodeGemma Team. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Cunxiang Wang, Sirui Cheng, Qipeng Guo, Yuanhao Yue, Bowen Ding, Zhikun Xu, Yidong Wang, Xiangkun Hu, Zheng Zhang, and Yue Zhang. 2024. Evaluating open-qa evaluation. *Advances in Neural Information Processing Systems*, 36.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code representation learning at scale. *ArXiv*, abs/2402.01935.

[11] https://huggingface.co/bigcode/starcoder

[12] https://huggingface.co/codellama/CodeLlama-7b-hf

A Appendix

A.0.1 Approximate nearest neighbor in embeddings similarity ranking experiment

Faiss from 4.1 [13] is a library for efficient similarity search and clustering of dense vectors. It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM. It also contains supporting code for evaluation and parameter tuning. For the efficiency of raising such an index with large vectors, consider using an approximate nearest neighbor (ANN) algorithm. ANN algorithms can be more efficient than exhaustive search methods when dealing with high-dimensional data, especially in large vector spaces

Approximate Nearest Neighbor (ANN) approach was used to speed up the search process through a large number of data.

It's the approach that helps to group a large set of vectors by parts using the k-means algorithm, each part corresponding to a centroid - a vector that is the selected center for this cluster. It searches through the minimum distance to the centroid and then searches for the minimum distance among the vectors in that cluster corresponding to that centroid. Taking k equal to $\sqrt{n}$, where n is the number of vectors in the index, we obtain an optimal search at two levels - first among $\sqrt{n}$ centroids, then among $\sqrt{n}$ vectors in each cluster. The search is times faster than the full one, which solves one of the problems when dealing with millions of vectors.

Thus, instead of a complete search of vectors to find the nearest one, we need to find only the necessary cluster and search for vectors already in it.

A.1 RealCode benchmark details

The sources of the repositories used in creating the RealCode benchmark are:

- https://github.com/Jakob-98/openai-functools
- https://github.com/biobootloader/mentat
- https://github.com/causalens/cai-causal-graph
- https://github.com/modelscope/modelscope-agent
- https://github.com/simonmesmith/agentflow
- https://github.com/defog-ai/sql-eval
- https://github.com/Wyvern-AI/wyvern
- https://github.com/danielbeach/tinytimmy
- https://github.com/a-r-r-o-w/stablefused
- https://github.com/langchain-ai/permchain
- https://github.com/NullPyDev/beholder
- https://github.com/opencopilotdev/opencopilot
- https://github.com/AgentOps-AI/agentops
- https://github.com/TengHu/ActionWeaver
- https://github.com/fynnfluegge/doc-comments.ai
- https://github.com/Tinny-Robot/DimSense
- https://github.com/mljar/plotai
- https://github.com/juliendenize/eztorch
- https://github.com/yihong0618/epubhv
- https://github.com/simonw/llm-cluster
- https://github.com/Pennyw0rth/NetExec
- https://github.com/Vaultexe/server

We didn't apply any specific topic-based filtering to these data. The topic meta information comes from the topic distribution of Python repositories on GitHub in the summer of 2023. The repositories were rolled back to a specific commit during data preparation.

A.2 Snippet length variety experiment

We have examined submitting code snippets of different lengths. We attempt two options — to submit a small relevant piece of code or to submit the entire code from the file and then select the best snippet after ranking by BM25 4.2. The best snippet was fed into the LLMs input in the format shown in the input section (see 4 for reference).

We've examine submitting different lengths of code snippets. There were two alternatives - to feed a small piece of code that is relevant or to feed the whole code from the file from which the best snippet after ranking by BM25 4.2 was found. We've tried each of them, and the results are presented below.

The best snippets was feeded to the LLms input in the format presented in 4.

[13] https://github.com/facebookresearch/faiss

Model	CS	EM
DeepSeek Coder$_{snippet}$	0.84	0.74
DeepSeek Coder$_{whole\_file}$	0.84	0.73
CodeGemma$_{2B\,snippet}$	0.75	0.25
CodeGemma$_{2B\,whole\_file}$	0.75	0.24
CodeGemma$_{7B\,snippet}$	0.83	0.31
CodeGemma$_{7B\,whole\_file}$	0.82	0.30
CodeQwen$_{7B\,snippet}$	0.83	0.74
CodeQwen$_{7B\,whole\_file}$	0.83	0.75

Table 3: Snippet length variety experiment by BM25 ranking by RealCode benchmark. Best model scores are bold.

A.3 LLMs evaluation by nine programming languages with BM25 snippet codes ranking

We also conducted an evaluation of various models using the specified approach across nine programming languages. The results are documented in 1. This method consistently demonstrates an enhancement across all languages. The data was collected by the same concept as the RealCode benchmark A.1. We took projects from GitHub on actual code for nine different programming languages: Python, Java, JavaScript, TypeScript, C++, Go, C#, Kotlin, Ruby.

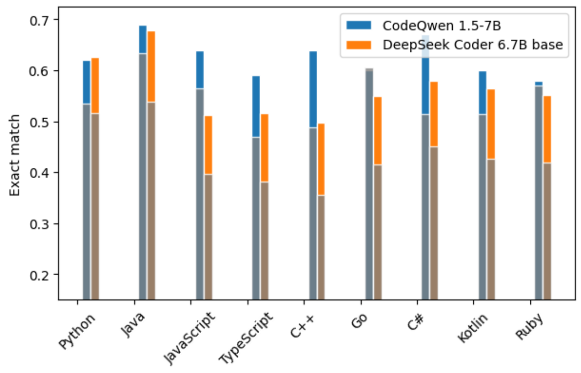

Figure 1: EM metric on the BM25 experiment on dataset gathered by 9 languages from GitHub projects.