Code Needs Comments: Enhancing Code LLMs with Comment Augmentation

Demin Song*[1], Honglin Guo*[1,2], Yunhua Zhou[1], Shuhao Xing[1,2]
Yudong Wang[1], Zifan Song[1], Wenwei Zhang[1]
Qipeng Guo†[1], Hang Yan[1,3], Xipeng Qiu[2], Dahua Lin[1,3]
[1]Shanghai AI Laboratory, [2]School of Computer Science, Fudan University
[3]The Chinese University of Hong Kong
{songdemin,zhouyunhua,xingshuhao.dispatch,wangyudong}@pjlab.org.cn
{songzifan,zhangwenwei,guoqipeng,yanhang,lindahua}@pjlab.org.cn
{hlguo20,xpqiu}@fudan.edu.cn

Abstract

The programming skill is one crucial ability for Large Language Models (LLMs), necessitating a deep understanding of programming languages (PLs) and their correlation with natural languages (NLs). We examine the impact of pre-training data on code-focused LLMs' performance by assessing the comment density as a measure of PL-NL alignment. Given the scarcity of code-comment aligned data in pre-training corpora, we introduce a novel data augmentation method that generates comments for existing code, coupled with a data filtering strategy that filters out code data poorly correlated with natural language. We conducted experiments on three code-focused LLMs and observed consistent improvements in performance on two widely-used programming skill benchmarks. Notably, the model trained on the augmented data outperformed both the model used for generating comments and the model further trained on the data without augmentation.

1 Introduction

The development of Large Language Models (LLMs) has made remarkable strides across various domains, including the field of code understanding and generation. Works such as CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023a), and Code Llama (Rozière et al., 2023a) have achieved significant breakthroughs in the task of natural language to code (NL2Code). Moreover, aligning natural language descriptions with their corresponding execution code to expand code-related training corpus to further enhance the model's coding capabilities has become a research focus for scholars (Yin et al., 2018; Ahmad et al., 2021; Wang et al., 2021b; Neelakantan et al., 2022; Muennighoff et al., 2023; Chai et al., 2023). Code Llama (Rozière et al., 2023a), which is currently one of the most popular code LLMs, also mentioned that 8% of their sample data was sourced from natural language datasets related to code. In fact, comments are the natural language components that are inherently related to code. Guo et al. (2022) had conducted ablation experiments to demonstrate that training models on code data with comments leads to improved ability. Moreover, the textbook and exercise data proposed by Gunasekar et al. (2023a), which is considered a prior work in the field of code LLMs, can be considered a form of comment in a sense. However, generating a large amount of such data using GPT is infeasible due to cost considerations.

Language	#Chars of Comment	#Chars	Comment Density
C#	5.4B	30.8B	0.1764
C++	6.6B	38.0B	0.1753
Go	3.0B	19.6B	0.1553
Java	12B	66.8B	0.1917
JavaScript	6.3B	46.9B	0.1352
PHP	5.1B	42.3B	0.1207
Python	9.6B	44.1B	0.2187
Ruby	0.9B	5.18B	0.1821
Rust	1.1B	6.44B	0.1641
TypeScript	2.4B	20.1B	0.1207
Average	**5.3B**	**32.0B**	**0.1670**

Table 1: Comment density across ten mainstream programming languages in StarCoder (Li et al., 2023a). #Chars of Comment indicates the number of non-white characters of the code comment. #Chars is the total number of non-white characters. In fact, high quality repositories even have comment density exceeding 40%, such as the case of mini redis[1]. This suggests that the existing code dataset indeed contains too few comments.

Considering that the alignment between natural language and code has not yet been relatively explored, comments serve as a representative and crucial bridge between the two. Therefore, the primary objective of this work is to explore the significance of comments. An intuitive supposition posits that an augmentation in training corpus that aligns code

* Equal contribution.
† Corresponding author

[1]https://github.com/tokio-rs/mini-redis

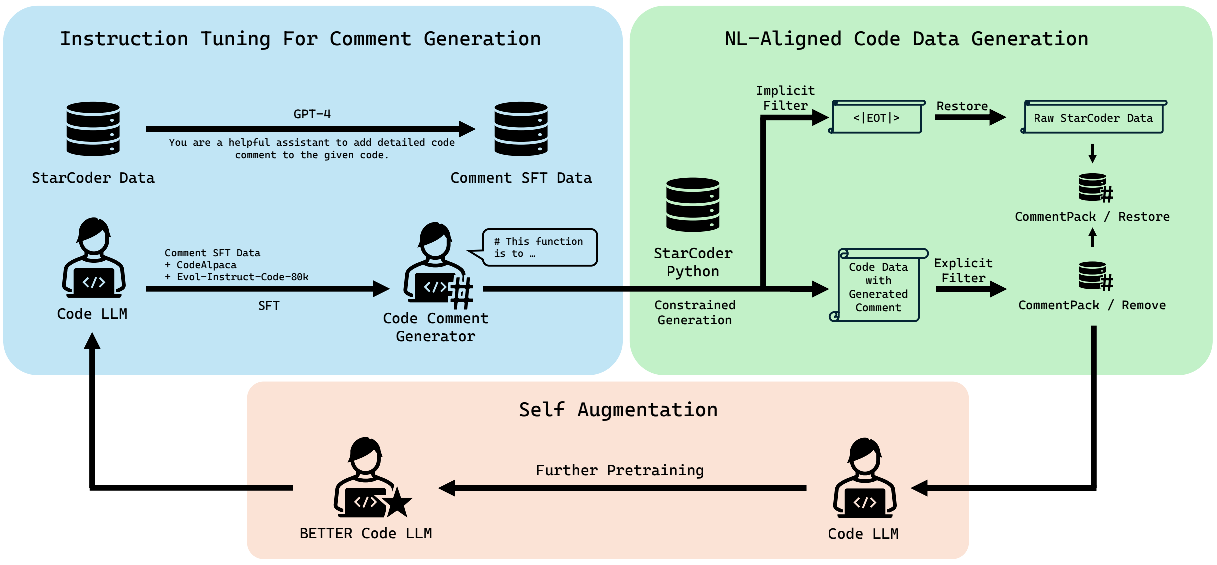


Figure 1: Illustrates the workflow of our proposed self-augmentation method. Firstly, it enables LLMs to generate comments for code through instruction tuning. Then, LLMs generate comments for existing code. The further training is conducted on enriched code data with comments, aiming to achieve self-augmentation.

and natural language (comments) will invariably enhance the model's performance. To quantify this alignment, we initially delineate "comment density" as the ratio of the number of non-white characters in comments to the total number of non-white characters and then examine how different levels of comment density impact downstream tasks.

As shown in Table 1, existing comments in code are limited. This severely hinders our goal of improving model performance and training efficiency by increasing the amount of aligned corpus between code and natural language. Therefore, we propose a novel method aimed at generating more aligned data, which is characterized by utilizing the powerful generation capabilities of LLMs to generate comments for the original code data. To accomplish this, we require a model capable of understanding code and providing corresponding comments. From this perspective, our method can also be viewed as a form of specialized data distillation. While, unlike traditional data distillation methods that rely on a teacher model, our approach accomplishes knowledge distillation through self-supervision. This represents the key distinction between our method and existing data distillation techniques. Table 2 provides detailed information on existing works.

To ensure that the code remains unchanged during LLMs generation and accelerate the generation process, we propose a constrained generation approach that generates data on a line-by-line basis, thereby circumventing the procedure of LLMs deleting, modifying the original code or producing new code. Considering the need to exercise caution in trusting the comments added by the model, we introduce a discriminator in this study to filter out extreme cases. The discriminator evaluates the generated comments and filters out samples that exhibit significant differences from the original code. In our experiments, we observe that utilizing LLMs for comments generation not only enhances the capabilities of the base model but also facilitates self-augmentation. The overall framework of this work is depicted in Figure 1

We highlight our contributions as follows:

- We discovered that the density of comments in pre-training code significantly affects the performance of LLM models in downstream tasks, and based on this, we proposed a new data augmentation method.
- We introduced a new inference method for generating comments, forming an efficient self-augmentation pipeline.
- Our method achieved substantial improvements on Llama 2, Code Llama, and InternLM2.

2 Related Work

2.1 Alignment between Code and Natural Language

Yin et al. (2018) proposed the effective uti-

Models	SFT	Pre-Training	Natural Language	Code	Samples	Tokens
phi-1(Gunasekar et al., 2023b)	✓	✓		✓	-	1B
WizardCoder(Luo et al., 2023a)	✓			✓	78K	-
WaveCoder(Yu et al., 2023)	✓			✓	20K	-
phi-1.5(Li et al., 2023b)		✓	✓		-	20B
WizardLM(Xu et al., 2023)	✓		✓		250K	-
Genie(Yehudai et al., 2024)		✓	✓		300K	-
Self-Instruct(Wang et al., 2023)	✓		✓		82K	-
Ours		✓		✓	6.5M	15.2B

Table 2: Existing data distillation methods rely on a teacher model to acquire knowledge, and are limited by the amount of available data.

lization of highly correlated Natural Language-Programming Language (NL-PL) pairs to enhance the capabilities of code models in tasks such as code retrieval, summarization, and generation. Ahmad et al. (2021) employed Denoising Pre-training to establish semantic relationships between natural language and code, resulting in promising outcomes. Similarly, Wang et al. (2021b) focused on aligning natural language and code by incorporating NL2Code and Code2NL generation tasks into the pre-training phase. Neelakantan et al. (2022) achieved superior performance over CodeBERT in the code retrieval task by employing contrastive learning to align code and natural language. Muennighoff et al. (2023) enhanced the code model's ability to generate code that follows natural language by utilizing commit messages.

The significance of comments as a component inherently related to code has also garnered considerable interest in research. Feng et al. (2020) employed the Masked Language Modeling (MLM) task on code data with comments to train a pre-trained model, yielding excellent results. Wang et al. (2021a), on the other hand, utilized Contrastive Learning to align code with comments. Furthermore, Guo et al. (2022) conducted ablation experiments to demonstrate that training models on code data with comments leads to improved outcomes. In order to align natural language (NL) and code, Christopoulou et al. (2022) conducted a two-stage training specifically on the pairs of NL-code. This approach resulted in a significant performance improvement of approximately 70% compared to the single-stage training. While PL-NL alignment is of paramount importance, it is challenging to obtain naturally aligned data at the scale required for pre-training purposes.T herefore, we employ LLMs to generate corresponding natural language expressions based on the existing code.

2.2 Data Augmentation in the Field of Code

Code augmentation techniques can be categorized into Rule-based Techniques and Model-based Techniques. Rule-based methods often involve techniques such as replacing variable names, renaming method names, and inserting dead code to transform code snippets. Some code transformations also consider deeper structural information, such as control-flow graphs (CFGs) and use-define chains (UDCs) (Quiring et al., 2019). Model-based Techniques commonly utilize pre-trained models to replace non-keywords in the original data (Song et al., 2022). Another approach employed is similar to Back-Translation, where code translation tasks are augmented by translating between two programming languages using natural language as an intermediate language (Sennrich et al., 2016).

In addition, there are also several methods based on Example Interpolation Techniques. For instance, Dong et al. (2022) merged rule-based techniques for source code models with mixup to blend the representations of the original code snippet and its transformed counterpart. Li et al. (2022) introduces two novel interpolation techniques, namely Binary Interpolation and Linear Extrapolation, for source code models. Diverging from the aforementioned approach, we present a novel methodology as the pioneering endeavor to enhance comments by leveraging existing code.

2.3 Data Distillation in the Field of LLMs

In this work, our approach of data augmentation through the utilization of LLMs can be regarded as a form of data distillation. Such tasks typically rely on two processes: generation and filtering. Unnatural Instructions and Self-Instruct (Honovich et al., 2023; Wang et al., 2023) have employed this method in the creation of an instruction dataset. While following the aforementioned two steps, WizardLM and WizardCoder (Xu et al., 2023; Luo

Language	C#	C++	Go	Java	JavaScript
Instruct Num	447	364	425	435	458
Language	Python	PHP	Ruby	Rust	TypeScript
Instruct Num	495	449	466	391	462

Table 3: We constructed over 4000 instruction data from a total of 10 mainstream code of StarCoder (Li et al., 2023a).

et al., 2023a) utilized an Instruction Evolver to generate more diverse data. In fact, as the competency of the Teacher model has advanced, numerous studies have gradually phased out the step of using a discriminator to filter data (Gunasekar et al., 2023b; Li et al., 2023b).

However, the data generated by these methods all originates from the Teacher model, which often limits them to the knowledge of the Teacher. To mitigate this limitation, GENIE (Yehudai et al., 2024) proposes generating task-specific examples from the content. Similarly, in WaveCode (Yu et al., 2023), the code generation task involves generating instructions from code. Taking a step further, our method completely liberates itself from the constraints of a teacher model, enabling highly efficient generation of large-scale pre-training data.

3 Method

Indeed, generating comments for existing code by using LLMs is not a simple task for us with two principal challenges. Firstly, LLMs often struggle to effectively follow the "add comments" instruction, resulting in code loss or insufficient comment additions, especially for longer code files. Secondly, generating comments for large-scale pre-training code data can be computationally expensive, leading to significant training costs for the entire model. Appendix A is a bad case where LLMs fail to follow the instruction of "add comments".

3.1 Instruction Tuning for Comment Generation

In order to endow LLMs with the capacity to rigorously follow "add comments" instructions, we deliberately constructed an Instruction dataset for fine-tuning LLMs.

Instruction Dataset In this work, we selecte over 4000 samples from the 10 distinguished programming languages discussed in StarCoder Datasets (Li et al., 2023a). These samples were then augmented with corresponding comments using the GPT-4 model (OpenAI, 2023), resulting

~~~
Prompt: Please add detailed comments to the following code:
```python
from ..remote import RemoteModel
class NetworkDevicesGridRemote(RemoteModel):
    properties = ("id",
                  "DeviceID",
                  "DeviceIPDotted",
                  "DeviceName",
                  "DeviceType",
                  )
```

Output: ```python\n<|EOT|>\n```
~~~

Figure 2: A code of such nature, which is uncomplicated and composed solely of a single class definition, is deemed as low-quality training data. Consequently, we designate the output of this instructional data as <|EOT|>. This leads to the scenario where, upon fine-tuning an LLM with this variety of instruction data, the LLM gains the competency to discern such inferior quality data.

in the creation of an extensive instruction dataset. Following a meticulous manual screening process, we refined the dataset, retaining a total of 4394 high-quality instruction data instances. Then, we convert the prompt and code into Markdown format. Please find the sample of our instruction data from Appendix B

To mitigate the risk of the model overfitting to the specific characteristics of the instruction data, we incorporated additional datasets: CodeAlpaca (Chaudhary, 2023) and Evol-Instruct-Code-80k (Luo et al., 2023b). To ensure the uniqueness of our instructions, we meticulously removed any instruction data with comments that overlapped with the CodeAlpaca and Evol-Instruct-Code-80k datasets. After creating instruction data, we use it to finetune our base model: CodeLlama-7b (Rozière et al., 2023a) and obtain a code comments generator.

For a comprehensive overview of the language distribution within our instruction dataset for comment generation, please refer to Table 3

Implicit Filter Although the StarCoder (Li et al., 2023a) dataset underwent certain filtering processes, there are still some data instances that lack training value (e.g., containing only module imports, version specifications, or very simple class definitions). To counteract this predicament, we incorporated particular samples within the instruction datasets, wherein the output was designated as "<|EOT|>" to signify that the model does not deem the input code is worth adding comments. This strategy is designed with the objective of endowing the model with the capacity to recognize high-quality code data throughout the process of comments generation. Figure 2 provides an example of such a sample. Please refer to Appendix C for more details.

3.2 NL-Aligned Code Data Generation

To ensure the preservation of the original code during the comments generation process and to facilitate a degree of acceleration, we introduce a novel method of constrained generation. Indeed, preservation of the original code is crucial to avoid the model generating illusory, repetitive code. Further details and information regarding this aspect can be found in the Appendix D

Constrained Generation In the task of generating comments for existing code, there is a notable characteristic in the LLM's decoding stage: the generated content of the model can be easily separated into comments and code on a line-by-line basis. Since the code is precisely the input given to the model, we can directly skip the process of generating code by the model.

More formally, let $C = \{C_i\}$ represent the code data for which comments are to be generated, where C_i denotes the i-th line of the code. Let $x = \{\text{prompt}, C\}$ be the input sequence, and y_t^l be the t-th token generated by the LLM in the l-th line. It is worth noting that this generation process is performed on a line-by-line basis.

$$y_t^l \sim \begin{cases} P(y|x, y^{<l}, y^l_{<t}) & y^l_{<t} \text{ is comment,} \\ C_j & y^l_{<t} \text{ is code.} \end{cases} \quad (1)$$

In fact, during the process of generating each line of data of LLMs, it is possible to determine whether a particular line is code or not by using regular expressions with just a few initial tokens.

Please refer to Algorithm 1 for the pseudo code and Figure 3 for an illustration of our method.

Explicit Filter To exclude exceedingly poor instances in the comments generated by LLMs and ensure the quality of generated comments, we introduce two additional filtering rule:

- Excluding code data generated by LLMs that does not adhere to the markdown format.
- Excluding code data generated by LLMs where the discrepancy in length between the generated code and the original code exceeds 100%.

Algorithm 1: Constrained Generation Algorithm

```
Input  : $x$: Input sequence
         $C = \{C_1, \dots, C_n\}$: List of code
Output : $y$: Generated sequence
1 $y \leftarrow []$ ; /* Initialize empty output sequence */
2 while true do
3     $o \leftarrow$ LLM$(x, y)$ ;  /* Generate next output token using LLM */
4     if not gen_code (y, o) then
5         APPEND $(y, o)$ ;  /* If not generating code, append token to output */
6     else
7         EXTEND $(y,$ POP $(C))$ ;  /* If generating code, extend output with next code constraint */
8     if stop $(y)$ then
9         break ;  /* Break loop if stopping condition is met */
```

3.3 Self Augmentation

Upon executing the aforementioned two processes, we will acquire a high-quality code dataset with extensive comments. We can then proceed to conduct additional training to augment the capabilities of our base model, resulting in a better code LLM. This process engenders a self-augmentation feedback loop. Subsequently, the better LLM will serve as the base code LLM for the next iteration of self-augmentation, to be performed repeatedly. The overall process of our approach is illustrated in Figure 1.

4 Experiments

We initially lay the foundation with empirical evidence on the Llama 2 model (Touvron et al., 2023), illustrating that the fortification of alignment between code and natural language—particularly through the amplification of comment density—profoundly influences downstream tasks. Subsequently, we apply our proposed methodology to the Code Llama model (Rozière et al., 2023b), underscoring its capacity not merely to bolster weak baselines such as Llama 2, but also to achieve self-augmentation on models like Code Llama, distinguished by their exceptional performance in code generation tasks. Moreover, we have substantiated through the InternLM2 (Team, 2023) which is the most recent state-of-the-art LLm in the field. that the PL-NL alignment data, generated by CodeLLama, retains its efficacy for other models. All models were validated on the Hu-

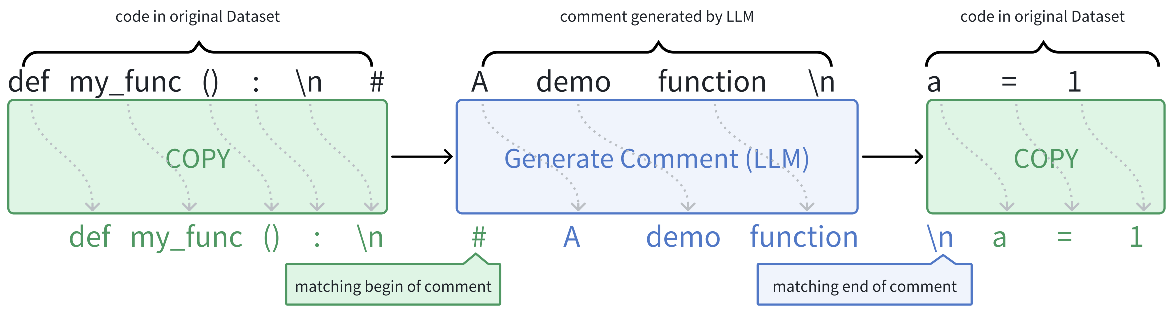


Figure 3: Illustration of the constrained generation algorithm. During the generation process, the code will be directly copied into the output until it encounters the marker indicating the beginning of a comment (#, ”’ or """ for Python). The commented portion is generated by the code comment generator until the end of the comment (\n, ”’ or """, correspondingly).

manEval (Cobbe et al., 2021) and MBPP (Austin et al., 2021) datasets.

4.1 Dataset

As an initial step, we selected to utilize the Python data from StarCoder (Li et al., 2023a) as our experimental validation dataset, henceforth referred to as **SP** (StarCoder Python) to circumvent any potential confusion. Leveraging the instruct data formulated in the preceding section, we enacted instruct tuning on the CodeLlama-7b model, thereby equipping it with the capability to generate comments for code. This model was subsequently employed to append comments to the SP dataset.

Owing to the existence of code data in StarCoder, characterized by an excessive number of tokens, the procedure of incorporating comments frequently surpasses the model's maximum sequence length. Consequently, we opted to exclude this subset of data from the comment addition process, preserving it for subsequent datasets.

Within our approach, we integrated both implicit and explicit filters to ensure the integrity of the code data and the generated comments. As a result, a considerable proportion of data was unable to pass through the implicit filter (model outputting <|EOT|>) or the explicit filter during the comment generation process. We adopted two distinct strategies to address this situation:

- Discarding the data that failed to traverse the implicit or explicit filter, culminating in a superior-quality dataset labeled **CommentPack / Remove** (CP/Remove, remove <|EOT|> samples in comment-packed python data).

Dataset	#Samples	Comment Density (%)	#Tokens
StarCoder Python	12.8M	21.87	20.8B
StarCoder Python / Remove	6.54M	23.08	13.1B
StarCoder Python / Absent	12.8M	0.0	16.7B
CommentPack / Restore	12.8M	32.59	21.5B
CommentPack / Remove	6.54M	38.23	15.2B

Table 4: Number of samples, comment density and number of tokens of the corresponding code datasets.

- Substituting the model's output with the original code data for instances that were unable to pass through either filter, leading to a lower-quality dataset (maintaining the same scale as the original dataset), designated as the **CommentPack / Restore** (CP/Restore, substitute raw StarCoder data for <|EOT|> samples in comment-packed python dataset) dataset.

Moreover, to streamline comparisons with the CP/Remove dataset, we gathered the corresponding original data for these instances, thereby constructing the **StarCoder Python / Remove** (SP/Remove, remove <|EOT|> samples in original python dataset of StarCoder) dataset.

In addition, to validate the importance of comments in the code dataset, we utilized regular expressions to eliminate all comments from the SP dataset, thus creating a pure code dataset. This dataset solely consists of code samples without any accompanying comments, named **StarCoder Python / Absent** (SP/Absent, means the absence of comments in the python dataset of StarCoder) Table 4 provides a detailed overview of the datasets mentioned.

4.2 Training Details

Further Training Our optimizer is AdamW (Loshchilov and Hutter, 2019) with β_1 and β_2 value

MODEL	DATA	HumanEval			MBPP		
		pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
Llama2-7b	-	12.25	19.75	23.73	20.81	29.10	37.75
	SP/Absent	16.46	27.87	34.22	19.00	40.10	48.16
	SP	17.07	31.09	**39.06**	20.40	**52.45**	**50.90**
	CP/restore	**23.17**	**31.79**	38.84	**29.20**	41.20	49.34
CodeLlama-7b	-	31.10	45.75	56.81	42.80	56.50	64.82
	SP	32.32	43.70	53.41	**45.00**	**58.03**	**65.41**
	SP/Remove	33.54	46.87	57.33	44.80	57.68	65.23
	CP/Restore	32.32	47.81	57.27	44.20	57.10	64.97
	CP/Remove	**39.02**	**51.89**	**61.50**	43.00	56.70	64.97
InternLM2-7b-base	-	32.32	49.64	60.13	41.40	54.06	62.23
	SP	35.98	49.82	59.57	43.00	56.24	64.18
	CP/Remove	**40.20**	**50.90**	**60.78**	**43.00**	**56.87**	**64.99**
InternLM2-7b	-	43.29	56.31	67.64	44.00	57.72	63.10
	SP	42.70	**59.67**	**70.72**	42.60	61.61	67.18
	CP/Remove	**49.39**	58.04	68.27	**47.80**	**64.89**	**71.12**

Table 5: Experiment results of further pre-training. "-" indicates the origin model without tuning. Almost all of the base models achieved leading performance on dataset SC/Remove, especially in the results of Pass@1.

of 0.9 and 0.95. We use a cosine scheduler with 250 warm-up steps, and set the final learning rate to be 1/10 of the peak learning rate. We use a batch size of 4M tokens which are presented as sequences of 4,096 tokens for Llama 2, 16384 tokens for Code Llama and InternLM 2. 40B tokens in total. We set the initial learning rate to $1e^{-5}$ for Llama 2, $3e^{-6}$ for Code Llama and InternLM2.

Instruction Training To further assess the performance of our model, we conducted instruction tuning using the dataset proposed by Alchemist-Coder(Song et al., 2024). The training was performed with a batch size of 512K tokens, organized as sequences of 8192 tokens. We employed a learning rate of $1e^{-5}$ and trained the model for 2 epochs on a cluster consisting of 32 NVIDIA A100-80GB GPUs.

4.3 Data Distillation

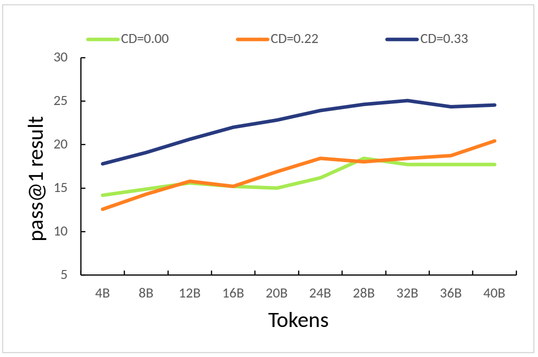
Table 5 shows the experimental results conducted on the Llama2-7b model. The results clearly demonstrate that as the comment density increases (with a comment density of 0 for “SP/Absent” and a density of 38.23% for “CP/Remove”), the model’s performance exhibits significant improvements transitioning from 16.46 to 23.17 on HumanEval dataset, 19.00 to 29.20 on MBPP dataset.

From Figure 6(a), it is clear that when training with the same number of tokens, data with a higher comment ratio achieves better results in downstream tasks. This result indicates that, under the same amount of data, a higher comment density makes it easier to learn the code, improves the alignment between natural language and code, and is more beneficial for code generation-oriented downstream tasks

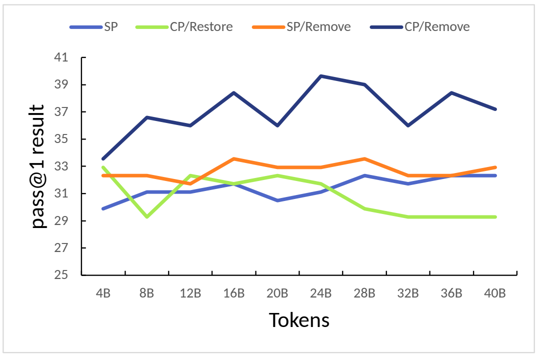
4.4 Self-Augmentation

Firstly, Table 5 provides a comprehensive overview of the results obtained from Further Training of Code Llama on the SP and CP/Restore datasets. The analysis reveals that merely replacing the filtered data, removed by explicit and implicit filters, with the original data does not significantly improve the model’s performance on downstream tasks. However, when the filtered data is completely removed (as observed in Code Llama’s results on SP and SP/Remove), a certain degree of improvement can be observed on the HumanEval evaluation set. Although this improvement may not be substantial, it still underscores the necessity of the filters. Similar conclusions can be drawn from the comparison of Code Llama’s further training results on CP/Restore and CP/Remove datasets.

For the same filtered data, the addition of more comprehensive comments leads to significant performance gains on HumanEval after further training (as evident from Code Llama’s results on CP/Remove and CP/Restore). However, it should be acknowledged that the structure of MBPP’s data



(a) Result of further pre-training on Llama 2 7B, CD means Comment Density



(b) Result of further pre-training on Code Llama 7B

Figure 4: HumanEval performance variation with respect to the number of training tokens.

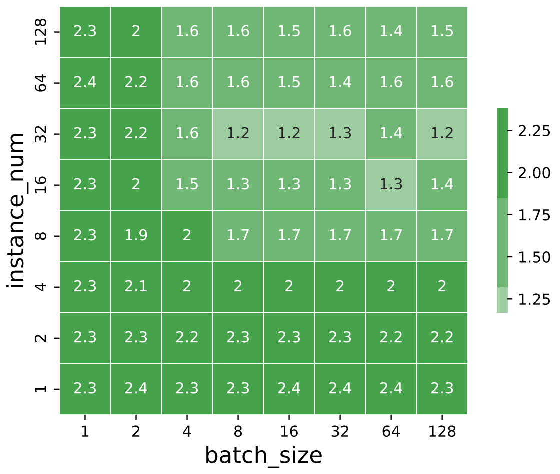
MODEL	DATA	HumanEval	MBPP
CodeLlama-7b	-	63.40	53.20
CodeLlama-7b	SP	**66.46**	55.80
CodeLlama-7b	CP/Remove	65.85	**58.60**

Table 6: Experiment Pass@1 result in HumanEval and MBPP of Instruction Fine-tuning."-" indicates the origin model without tuning.

and the way we incorporate data into the code differ significantly, and we did not achieve substantial improvements during the further training phase on MBPP. Nevertheless, we discovered that this does not imply a lack of substantial performance enhancement for the model. In fact, as show in Table 6, when Code Llama undergoes instruction tuning after further pre-training on SP and CP/Remove datasets, it further enhances the model's adaptability to the MBPP dataset, resulting in a noteworthy improvement of 5.4% pass@1 on CP/Remove. Please refer to the Appendix E for the results of Pass@5 and Pass@10.

Furthermore, the comment generated by our approach on Code Llama remain effective for other models as well (as demonstrated by the comparison with further training results on SP and CP/Remove of InternLM2, where Code Llama's comments yield a significant improvement of 6% pass@1 on HumanEval for the InternLM2-7b-base model, 6.6% pass@1 on HUmanEval, 5.2% pass@1 on MBPP for the InternLM2-7b model).

Lastly, Figure 6(b) demonstrates that the data quality of SP/Remove surpasses that of SP. Furthermore, after incorporating comments into SP/Remove (CP/Remove), there is a significant qualitative improvement in the dataset's quality. This leap in data quality can be observed if we acknowledge the close correlation between data quality and downstream tasks, under the assumption that the base model remains consistent.



instance_num \ batch_size	1	2	4	8	16	32	64	128
128	2.3	2	1.6	1.6	1.5	1.6	1.4	1.5
64	2.4	2.2	1.6	1.6	1.5	1.4	1.6	1.6
32	2.3	2.2	1.6	1.2	1.2	1.3	1.4	1.2
16	2.3	2	1.5	1.3	1.3	1.3	1.3	1.4
8	2.3	1.9	2	1.7	1.7	1.7	1.7	1.7
4	2.3	2.1	2	2	2	2	2	2
2	2.3	2.3	2.2	2.3	2.3	2.3	2.2	2.2
1	2.3	2.4	2.3	2.3	2.4	2.4	2.4	2.3

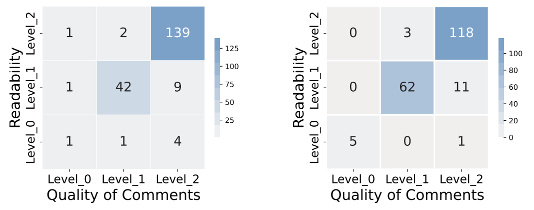
Figure 5: Heat map of speedup ratio across different combinations of instance numbers and batch sizes.

4.5 Constrained Generation

We have implemented the Constraint Generation method on LMDeploy[2] and demonstrated its effectiveness in accelerating decoding under different experimental. Despite LMDeploy already incorporating various acceleration techniques such as page attention, our method exhibits notable speed improvements.

As evident from Figure 5, the results indicate that our method achieves the most significant acceleration when the batch size and instance number are relatively small. Even when the GPU is oper-

[2] https://github.com/InternLM/lmdeploy



(a) Data Quality Before and After Adding Comments (b) Data Quality Before and After modifying Comments

Figure 6: Heat map of data quality. Level_0 means original data is better, Level_1 indicates a comparable level of quality, Level_2 signifies data with added comments is better

ating at maximum capacity (e.g., batch_size=128, instance_num=128), our method still provides a certain degree of speed enhancement.

4.6 Quality Control for Pre-training Data

To illustrates quality of the pre-training data, we introduce a new set of experiments to demonstrate the quality of comments generated by the model.

Comparing Data Quality Before and After Adding Comments : Randomly selecting 200 code data instances that originally had no comments, we conducted manual annotation to compare the readability of the code data and the accuracy of the added comments after applying the "add comments" process.

Comparing Data Quality Before and After modifying Comments : Randomly selecting 200 code data instances that originally had comments, we conducted manual annotation to compare the readability of the code data and the accuracy of the added comments after applying the "add comments" process.

Figure 6 demonstrate that the quality control methods proposed in our paper are indeed effective. They can even improve code quality to some extent.

5 Conclusion

In this paper, we propose a novel method of code data augmentation that generates comments for existing code. We validate its effectiveness on three different LLMs. This signifies a novel paradigm shift towards self-augmentation for code LLMs, thereby illuminating the latent potential for LLMs to self-evolve and enhance.

6 Limitation

In this paper, although we have successfully eliminated the reliance on data distillation with a teacher model, it is important to note that performing data augmentation on the pre-training dataset still incurs considerable GPU overhead. Additionally, using "<|EOT|>" as the model's output in the implicit filter stage may not align well with the behavioral patterns typically exhibited by a language model. It might be more beneficial to consider using natural language instead. Furthermore, during the next iteration of self-augmentation, we observed only marginal improvements, which is why these results were not reported in the main experiments. Further exploration and investigation are needed in this regard.

Acknowledgements

This work is supported by Shanghai Artificial Intelligence Laboratory. Sincerely appreciate thethoughtful comments by the reviewers of our paper. Special thanks to QiuyinzheZhang, FanCui and all the annotators, whose invaluable efforts have greatly contributed to this work.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. pages 2655–2668.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2023. ERNIE-code: Beyond English-centric cross-lingual pretraining for programming languages. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10628–10650, Toronto, Canada. Association for Computational Linguistics.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. Pangu-coder: Program synthesis with function-level language modeling.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Zeming Dong, Qiang Hu, Yuejun Guo, Maxime Cordy, Mike Papadakis, Yves Le Traon, and Jianjun Zhao. 2022. Enhancing code classification by mixup-based data augmentation. *CoRR*, abs/2210.03003.

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. EMNLP 2020:1536–1547.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023a. Textbooks are all you need. *CoRR*, abs/2306.11644.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023b. Textbooks are all you need. *CoRR*, abs/2306.11644.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. pages 7212–7225.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2023. Unnatural instructions: Tuning language models with (almost) no human labor. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 14409–14428. Association for Computational Linguistics.

Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022. Exploring representation-level augmentation for code search. pages 4924–4936.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023a. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need II: phi-1.5 technical report. *CoRR*, abs/2309.05463.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023a. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023b. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124.

Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. 2022. Text and code embeddings by contrastive pre-training. *CoRR*, abs/2201.10005.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.

R OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2:13.

Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security*

Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019, pages 479–496. USENIX Association.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023a. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023b. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data.

Zifan Song, Yudong Wang, Wenwei Zhang, Kuikun Liu, Chengqi Lyu, Demin Song, Qipeng Guo, Hang Yan, Dahua Lin, Kai Chen, et al. 2024. Alchemistcoder: Harmonizing and eliciting code capability by hindsight tuning on multi-source data. *arXiv preprint arXiv:2405.19265*.

Zixuan Song, Xiuwei Shang, Mengxuan Li, Rong Chen, Hui Li, and Shikai Guo. 2022. Do not have enough data? an easy data augmentation for code summarization. In *13th IEEE International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2022, Beijing, China, November 25-27, 2022*, pages 1–6. IEEE.

InternLM Team. 2023. Internlm: A multilingual language model with progressively enhanced capabilities. `https://github.com/InternLM/InternLM`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. pages 8696–8708.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *CoRR*, abs/2304.12244.

Asaf Yehudai, Boaz Carmeli, Yosi Mass, Ofir Arviv, Nathaniel Mills, Assaf Toledo, Eyal Shnarch, and Leshem Choshen. 2024. Genie: Achieving human parity in content-grounded datasets generation. *arXiv preprint arXiv:2401.14367*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 476–486. ACM.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

A Bad Cases of Comment Generation by LLMs

~~~
Prompt: Please add detailed comments to
    the following code
```python
# https://www.hackerrank.com/challenges/
    save-the-prisoner
~~~

```
# this is too slow


def save(n, sweets, s):
    sweets -= 1
    while sweets:  # while we have candy
        if s == n:  # if we reach the
            end, reset
            s = 0
        else:
            s += 1  # increase the id
        sweets -= 1
    return s


t = int(input().strip())
for a0 in range(t):
    n, m, s = input().strip().split(' ')
    n, m, s = [int(n), int(m), int(s)]

    print(save(n, m, s))

'''
Model Output:
'''python
# This code is designed to solve the "
    Save the Prisoner" problem on
    HackerRank.
# The problem is as follows:

# There are N prisoners standing in a
    circle waiting to be executed. The
    executions occur around a circular
    table, and each execution takes
    exactly one minute. The executions
    are carried out starting from the
    prisoner at position 1 and moving
    clockwise until the last prisoner is
     executed. The executions are
    carried out such that each prisoner
    is guaranteed to be executed.

# Given N, the total number of prisoners
    , M, the number of executions per
    hour, and S, the position of the
    prisoner to be executed next, find
    the position of the prisoner to be
    executed in the Nth hour.

# The provided code is an implementation
     of a solution to this problem. It
    uses a while loop to simulate the
    executions of the prisoners,
    decrementing the number of remaining
     sweets with each execution.

# The code first decrements the number
    of remaining sweets by 1. Then, it
    checks if there are any remaining
    sweets. If there are, it continues
    the loop.

# If the prisoner to be executed next (S
    ) is equal to N, it means that the
    circle has returned to the beginning
    , so it resets S to 0. Otherwise, it
     increments S by 1 to move to the
    next position.

# Finally, it returns the value of S,
    which represents the position of the
     prisoner to be executed in the Nth
    hour.

# This code is too slow because it does
    not take into account the fact that
    the executions occur around a
    circular table, which means that the
     order of executions may not be in a
     strict linear sequence.
# A more efficient solution would be to
    consider the fact that each
    execution takes exactly one minute,
    and the circle returns to the
    beginning after M executions per
    hour.
# This would allow the code to predict
    the position of the prisoner to be
    executed in each hour, and thus find
     the position of the prisoner to be
    executed in the Nth hour.
'''
```

Listing 1: a bad case where LLMs fail to follow the instruction of "add comments".

B A Sample of Instuctions Data

```
Prompt: Please add detailed comments to
    the following code
'''python
import sys
import os
import glob

if len(sys.argv) != 2:
  print("Error: wrong format.\nUsage:
      python find_class.py [class_name]"
      )
  sys.exit(0)

searching_class_name = sys.argv[1]

def find_class(class_name):
  file_list = glob.glob('*.txt')
  file_list.sort()
  # iterate through the text files
  file_found = False
  for txt_file in file_list:
    # open txt file lines to a list
    with open(txt_file) as f:
      content = f.readlines()
    # remove whitespace characters like
        '\n' at the end of each line
    content = [x.strip() for x in
        content]
    # go through each line of eache file
    for line in content:
      class_name = line.split()[0]
      if class_name ==
          searching_class_name:
        print(" " + txt_file)
        file_found = True
        break
  if not file_found:
    print(" No file found with that
        class")
```

```

print("Ground-Truth folder:")
os.chdir("../ground-truth")
find_class(searching_class_name)
print("\nPredicted folder:")
os.chdir("../predicted")
find_class(searching_class_name)

‘‘‘
Model Output:
‘‘‘python
# import necessary libraries
import sys
import os
import glob

# check if the command line arguments are valid
if len(sys.argv) != 2:
    print("Error: wrong format.\nUsage: python find_class.py [class_name]")
    sys.exit(0)

# get the class name from the command line arguments
searching_class_name = sys.argv[1]


# function to find the class in the text files
def find_class(class_name):
    # get all text files in the current directory
    file_list = glob.glob("*.txt")
    file_list.sort()
    # flag to check if the class is found in any file
    file_found = False
    # iterate through the text files
    for txt_file in file_list:
        # open txt file lines to a list
        with open(txt_file) as f:
            content = f.readlines()
        # remove whitespace characters like ‘\n‘ at the end of each line
        content = [x.strip() for x in content]
        # go through each line of each file
        for line in content:
            class_name = line.split()[0]
            # if the class name matches the input class name
            if class_name == searching_class_name:
                print(" " + txt_file)
                file_found = True
                break
    # if the class is not found in any file
    if not file_found:
        print(" No file found with that class")


# find the class in the ground-truth folder
print("Ground-Truth folder:")
os.chdir("../ground-truth")
find_class(searching_class_name)

# find the class in the predicted folder
print("\nPredicted folder:")
os.chdir("../predicted")
find_class(searching_class_name)

‘‘‘
```

Listing 2: a sample of our instuctions data.

C Details of the Implicit Filter

The output of instruction data with the following types will be set as “<|EOT|>”:

- The code of the instruction data consists solely of comments.
- The code of the instruction data only contains package imports.
- The code of the instruction data only consists of simple class definitions.
- The code of the instruction data only contains variable definitions without any operations.
- The code of the instruction data does not have a function body.
- The code of the instruction data has a function body that is too simple and contains only a few operations.
- The code of the instruction data only consists of a large number of configuration definitions.

D Bad Cases of Original Generation

```
Prompt: Please add detailed comments to the following code
‘‘‘python
import functools
import logging


@functools.lru_cache
def component_logger(name: str):
    return logging.getLogger(name)

‘‘‘
Model Output:
‘‘‘python
import functools
import logging


# Decorator to cache the results of the function, using the function arguments as the cache key
```

~~~
# This will make the function faster by returning the cached result when the same inputs are provided
@functools.lru_cache
def component_logger(name: str):
    # This function is responsible for creating a logger object for a given component
    # The logger object is used to log messages for a specific component

    # Get the logger for the given component name
    # The logger is a singleton object that can be used to log messages for a specific component
    # The 'name' parameter is used to identify the component for which the logger is being created
    logger = logging.getLogger(name)

    # Set the log level to INFO
    # This is the default log level in Python logging module, which means that only log messages of level INFO or higher will be displayed
    logger.setLevel(logging.INFO)

    # Create a file handler to log messages to a file
    # This handler will write log messages to a file named 'component_name.log'
    # The log messages will be written in the 'INFO' level
    file_handler = logging.FileHandler(f"{name}.log")
    file_handler.setLevel(logging.INFO)

    # Create a formatter for the file handler
    # The formatter will format the log messages in a specific format
    formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")
    file_handler.setFormatter(formatter)

    # Add the file handler to the logger
    # The file handler will be used to write log messages to the file
    logger.addHandler(file_handler)

    # Return the logger object
    # This logger object can be used to log messages for the specified component
    return logger

```
~~~

Listing 3: a bad case where an LLM generates an illusion using the original generation approach.

~~~
Prompt: Please add detailed comments to the following code:
```python
#!/usr/bin/env python3

from mylib.genetackdb2 import Org, FSGene

class ChelOrg(Org):
    def __init__(self, gtdb, db_id):
        super().__init__(gtdb, db_id)
        self.load_prm(prm_str=[
        'chel_genotype_LMS',
        'chel_genotype_NHDI',
        'chel_genotype_NTS_HDI'
        ])

    def get_true_M_chelatase_fsgene_ids(self):
        return [d['id'] for d in self.gtdb.exec_sql_ar(
            '''select distinct fs.id from seqs s, fsgenes fs
            where s.org_id=%s and fs.seq_id=s.id and fs.fs_type <> 0
            ''', self.id)]

class ChelFSGene(FSGene):
    def __init__(self, gtdb, db_id):
        super().__init__(gtdb, db_id)
        self.load_prm(prm_str=['chel_gene', 'chel_subunit'])
```
Model Output :
```python
# This code defines two classes: ChelOrg and ChelFSGene.
# ChelOrg is a subclass of Org from the mylib.genetackdb2 module.
# ChelFSGene is a subclass of FSGene from the same module.

# The following line is a shebang that tells the system to use the specified interpreter to run the script.
# In this case, it's telling the system to use Python 3 to run the script.
# This is a Unix convention and doesn't apply to Windows systems.
# If the script is run on a Windows system, it will likely ignore this line.
#!/usr/bin/env python3

# The next line imports the Org and FSGene classes from the mylib.genetackdb2 module.
# These classes are likely used to interact with a database or perform other operations related to genetics.
from mylib.genetackdb2 import Org, FSGene

# The following class definition is for the ChelOrg class.
# It is a subclass of Org, which means it inherits all of the methods and properties of Org.
class ChelOrg(Org):
    # The __init__ method is a special method in Python that is
~~~

```
        automatically called when an object is created.
    # It is used to initialize the object with any necessary parameters or values.
    def __init__(self, gtdb, db_id):
        # The super().__init__ call is a way to call the __init__ method of the parent class (Org) from the child class (ChelOrg).
        # This allows the child class to inherit any functionality from the parent class.
        super().__init__(gtdb, db_id)
        # The load_prm method is likely used to load or set up any parameters needed by the class.
        # The prm_str parameter is a list of strings that specify the parameters to load.
        self.load_prm(prm_str=[
        'chel_genotype_LMS',
        'chel_genotype_NHDI',
        'chel_genotype_NTS_HDI'
        ])

    # This method returns a list of true M chelatase FSGene ids.
    # The method likely retrieves these ids from the database or performs some other operation.
    def get_true_M_chelatase_fsgene_ids(self):
        # The exec_sql_ar method is likely used to execute a SQL query against the database.
        # The query is a string that selects distinct FSGene ids from the database.
        # The '%s' placeholder is used to substitute the org_id with the current org_id of the object.
        return [d['id'] for d in self.gtdb.exec_sql_ar(
            '''select distinct fs.id from seqs s, fsgenes fs
            where s.org_id=%s and fs.seq_id=s.id and fs.fs_type <> 0
            ''', self.id)]

# The following class definition is for the ChelFSGene class.
# It is a subclass of FSGene, which means it inherits all of the methods and properties of FSGene.
class ChelFSGene(FSGene):
    # The __init__ method is a special method in Python that is automatically called when an object is created.
    # It is used to initialize the object with any necessary parameters or values.
    def __init__(self, gtdb, db_id):
        # The super().__init__ call is a way to call the __init__ method of the parent class (FSGene) from the child class (ChelFSGene).
        # This allows the child class to inherit any functionality from the parent class.
        super().__init__(gtdb, db_id)
        # The load_prm method is likely used to load or set up any parameters needed by the class.
        # The prm_str parameter is a list of strings that specify the parameters to load.
        self.load_prm(prm_str=['chel_gene', 'chel_subunit'])

# The code above defines two classes: ChelOrg and ChelFSGene.
# ChelOrg is a subclass of Org from the mylib.genetackdb2 module.
# ChelFSGene is a subclass of FSGene from the same module.

# The following line is a shebang that tells the system to use the specified interpreter to run the script.
# In this case, it's telling the system to use Python 3 to run the script.
# This is a Unix convention and doesn't apply to Windows systems.
# If the script is run on a Windows system, it will likely ignore this line.
#!/usr/bin/env python3

# The next line imports the Org and FSGene classes from the mylib.genetackdb2 module.
# These classes are likely used to interact with a database or perform other operations related to genetics.
from mylib.genetackdb2 import Org, FSGene

# The following class definition is for the ChelOrg class.
# It is a subclass of Org, which means it inherits all of the methods and properties of Org.
class ChelOrg(Org):
    # The __init__ method is a special method in Python that is automatically called when an object is created.
    # It is used to initialize the object with any necessary parameters or values.
    def __init__(self, gtdb, db_id):
        # The super().__init__ call is a way to call the __init__ method of the parent class (Org) from the child class (ChelOrg).
        # This allows the child class to inherit any functionality from the parent class.
        super().__init__(gtdb, db_id)
        # The load_prm method is likely used to load or set up any parameters needed by the class.
        # The prm_str parameter is a list of strings that specify the parameters to load.
        self.load_prm(prm_str=[
        'chel_genotype_LMS',
        'chel_genotype_NHDI',
        'chel_genotype_NTS_HDI
        '])

    # This method returns a list of true M chelatase FSGene ids.
    # The method likely retrieves these ids from the database or performs some other operation.
    def get_true_M_chelatase_fsgene_ids(self):
        # The exec_sql_ar method is likely used to execute a SQL query against the database.
        # The query is a string that selects distinct FSGene ids from the database.
        # The '%s' placeholder is used to substitute the org_id with the current org_id of the object.
        return [d['id'] for d in self.gtdb.exec_sql_ar(
            '''select distinct fs.id from seqs s, fsgenes fs
            where s.org_id=%s and fs.seq_id=s.id and fs.fs_type <> 0
            ''', self.id)]

# The following class definition is for the ChelFSGene class.
# It is a subclass of FSGene, which means it inherits all of the methods and properties of FSGene.
class ChelFSGene(FSGene):
    # The __init__ method is a special method in Python that is automatically called when an object is created.
    # It is used to initialize the object with any necessary parameters or values.
    def __init__(self, gtdb, db_id):
        # The super().__init__ call is a way to call the __init__ method of the parent class (FSGene) from the child class (ChelFSGene).
        # This allows the child class to inherit any functionality from the parent class.
        super().__init__(gtdb, db_id)
        # The load_prm method is likely used to load or set up any parameters needed by the class.
        # The prm_str parameter is a list of strings that specify the parameters to load.
        self.load_prm(prm_str=['
            chel_gene', 'chel_subunit'])
```
```
```

Listing 4: a bad case example of LLMs generating duplicate data. using the original generation approach.

E Experiment Result of Instruction Fine-Tuning

Table 7 presents the complete results of instruction fine-tuning on the Humaneval and MBPP datasets for Pass@1 to Pass@10

F Ethics Statement

We use OpenAI GPT to generate part of the training data. The terms of use can be accessed from OpenAI's official website[3].

We use CodeAlpaca and Evol-Instruct-Code-80k datasets for instruction tuning. They are distributed under CC-By-NC 4.0 license. You can get a copy of the licenses from their GitHub repositories[4].

We perform experiments using StarCoder as the validation dataset. The StarCoder dataset is distributed under Terms of Use for The Stack[5].

We employ Code Llama to generate comment. According to Code Llama's license[6], you will not use the Llama Materials or any output or results of the Llama Materials to improve any other large language model (excluding Llama 2 or derivative works thereof).

The experiments are performed on Llama 2, Code Llama and InternLM2. Their weights are distributed under their corresponding licenses[7].

Out of ethical considerations, we will release the CommentPack datasets and the further pre-trained model checkpoints only for research purpose under any relevant licenses.

[3]https://openai.com/policies/terms-of-use

[4]https://github.com/sahil280114/codealpaca/blob/master/DATA_LICENSE https://github.com/nlpxucan/WizardLM/blob/main/WizardCoder/DATA_LICENSE

[5]https://hf-mirror.com/datasets/bigcode/the-stack#terms-of-use-for-the-stack

[6]https://github.com/facebookresearch/codellama/blob/main/LICENSE

[7]https://github.com/facebookresearch/llama/blob/main/LICENSE https://github.com/facebookresearch/codellama/blob/main/LICENSE https://github.com/InternLM/InternLM#license

Model	DATA	HumanEval			MBPP		
		pass@1	pass@5	pass@10	pass@1	pass@5	passs@10
CodeLlama-7b	-	63.40	81.11	86.29	53.20	65.14	71.21
CodeLlama-7b	SP	66.46	80.91	86.46	55.80	65.60	71.25
CodeLlama-7b	CP/Remove	65.85	80.7	86.27	58.60	65.00	71.14

Table 7: Experiment results of instruction fine-tuning. Lines of DATA marked as "-" indicate the reported values of the origin model.