Semi-Instruct: Bridging Natural-Instruct and Self-Instruct for Code Large Language Models

Xianzhen Luo[1], **Qingfu Zhu**[1*], **Zhiming Zhang**[1],
Xu Wang[2], **Qing Yang**[2], **Dongliang Xu**[2], **Wanxiang Che**[1]
[1]Harbin Institute of Technology, Harbin, China,
[2]Du Xiaoman (Beijing) Science Technology Co., Ltd.,
{xzluo, qfzhu, zmzhang, car}@ir.hit.edu.cn,
{wangxu04, yangqing, xudongliang}@duxiaoman.com

Abstract

Instruction tuning plays a pivotal role in Code Large Language Models (Code LLMs) for the task of program synthesis. Presently, two dominant paradigms for collecting tuning data are natural-instruct (human-written) and self-instruct (automatically generated). Natural-instruct includes diverse and correct codes but lacks instruction-code pairs, and exists improper code formats like nested single-line codes. In contrast, self-instruct automatically generates proper paired data. However, it suffers from low diversity due to generating duplicates and cannot ensure the correctness of codes. To bridge the both paradigms, we propose **Semi-Instruct**. It first converts diverse but improper codes from natural-instruct into proper instruction-code pairs through a method similar to self-instruct. To verify the correctness of generated codes, we design a novel way to construct test cases by generating cases' inputs and executing correct codes from natural-instruct to get outputs. Finally, diverse and correct instruction-code pairs are retained for instruction tuning. Experiments show that semi-instruct is significantly better than natural-instruct and self-instruct. Furthermore, the performance steadily improves as data scale increases. Our code and data are public at [link].

Keywords: Program Synthesis, Large Language Model, Self-Instruct

1. Introduction

Program synthesis aims to generate code snippets given a specification, typically framed as a natural language description (Manna and Waldinger, 1980). It can effectively enhance programming efficiency and improve productivity. Meanwhile, the coding ability has been observed to be positively correlated with the performance of large language models (LLMs) on reasoning tasks (Shin and Van Durme, 2022; Yang et al., 2022; Chen et al., 2022), which further increases the attention to program synthesis. Similar to other generative tasks, the common practice of enhancing program synthesis ability is fine-tuning code LLMs on instruction-code pairs to align with human intentions (Wang et al., 2023b; Shen et al., 2023; Muennighoff et al., 2023). According to the source of instruction tuning data, the approaches for data collection can be further divided into two categories: **Natural-Instruct** (NI) (Mishra et al., 2022; Li et al., 2022) and **Self-Instruct** (SI) (Wang et al., 2023a; Luo et al., 2023).

NI aims at collecting human-written data from many code-related platforms such as GitHub and Codeforces. It consists of natural language such as extracted program comments or problem descriptions and corresponding codes or solutions. Two advantages lie in its ***diverse*** and ***correct*** (Zan et al., 2023) codes, as shown in the left part of Fig 1. On the one hand, such platforms offer massive diverse codes with distinct functionalities. On the other hand, the correctness of codes can be guaranteed by carefully controlling data sources such as selecting solutions from contest websites that have passed all human-created test cases. However, two drawbacks limit the performance of code LLMs tuning on NI data. First, ***improper*** coding formats such as nested single-line code (Bohnet and Döllner, 2011), and ambiguous variable names add too much noise to the data. Second, since natural languages always count only a small percentage (6.67%) (Ahmad et al., 2021) and most of them (sometimes formed as a single word such as "update") can not serve as instructions, the ***lack of instruction-code pairs*** which are high-quality and complete limits the program synthesis ability of code LLMs gained from instruction learning. Therefore, while NI provides massive diverse, and correct data, it is plagued by issues of improper coding formats and missing instruction-code pairs.

Conversely, SI leverages LLMs to automatically generate ***naturally paired*** (Taori et al., 2023) instructions and codes with ***proper*** format without human effort. However, its two shortcomings can not be ignored. First, the generated instruction-code pairs are of ***low diversity*** (Wang et al., 2023a) due to the constrained number of seed prompts. Second, absent from any testing or manual calibration, the correctness of generated codes is ***uncertain*** (Zhang et al., 2023b). Some approaches attempt to validate code by generating test cases consisting of inputs and outputs(Chen et al., 2023; Rozière et al., 2023). While the generated inputs

* Corresponding author

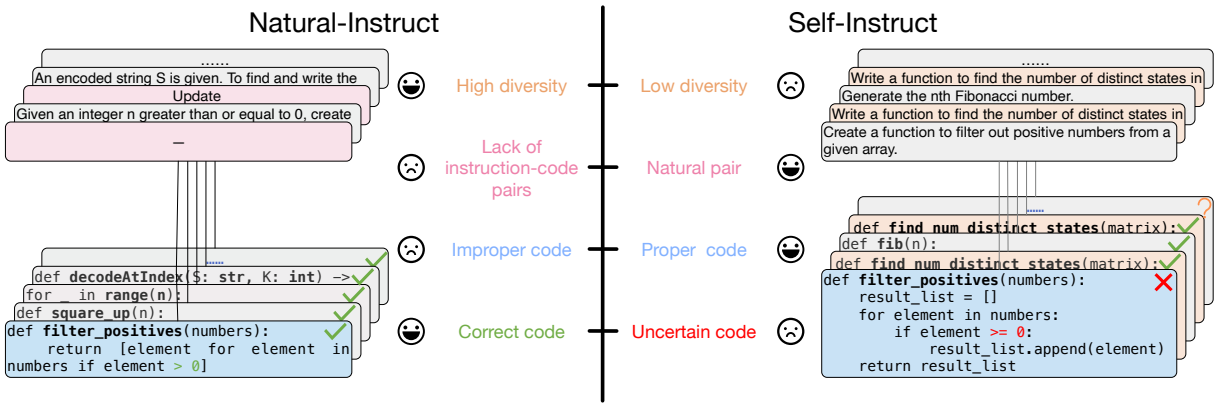

Figure 1: The advantages and disadvantages of natural-instruct and self-instruct. Their colors correspond to the data samples.

are mostly correct, we often fail to obtain expected outputs when faced with complex scenarios such as logical and numerical reasoning. Uncertain test cases not only waste generation resources but also are challenging to filter out. Thus, while SI excels in generating complete and clear instruction-code pairs, its limitations lie in the repetitive nature and inability to ensure correctness.

To this end, we introduce a novel approach, **Semi-Instruct** (SemI), to bridge the inherent strengths of both NI and SI. By leveraging the generative capability of LLMs like what SI does and feeding the diverse codes from NI (original code) into them, we can obtain the naturally paired instructions and proper codes (refined codes) that are correspondingly of high diversity. To validate the correctness of such refined codes, instead of directly generating the complete test cases at once as previous methods do, we offer a novel solution to handle it. Specifically, recall that it can always be guaranteed to obtain the correct inputs of test cases from LLMs, the corresponding correct outputs are supposed to be easily derived by executing the original codes, resulting in the more reliable test cases. Thus, these determined test cases function to affirm the accuracy of the refined codes.

The detailed process of SemI is divided into three steps: (1) **Generation**: Starting with an original code, LLMs generate an instruction, a fixed number of test cases' inputs based on the instruction, and a refined code in sequence. (2) **Validation**: Initially, we obtain outputs of test cases by running the generated inputs through the original code. However, since some inputs may cause runtime errors, not all of them result in complete test cases after execution. Then only the refined code that passes the remaining test cases is left. Finally, we eliminate any instructions that closely resemble previously generated ones. (3) **Ranking**: Intuitively, the quality of generated input depends on to what extent the LLMs understand the instruction. The more difficult the instruction is, the fewer test cases can be constructed after execution. Inspired by curriculum learning (Bengio et al., 2009), data are organized in descending order based on the count of test cases before tunning. **To the best of our knowledge, we are the first to get test cases consisting of generating inputs and executing outputs, and use the amount as a measure of difficulty.**

We carry out extensive experiments on the widely-used HumanEval dataset (Chen et al., 2021). When using only one type of data, SemI largely outperforms NI on each scale of data size and is also better than SI. Moreover, combining the data from SI and SemI outperforms SI alone by an average of 3% on p@1. Most importantly, the performance keeps steadily rising instead of oscillating or declining with the amount of data scale increasing.

The contributions are listed as follows:

- We propose a novel method named **Semi-Instruct**, which bridges the natural-instruct and self-instruct. Through semi-instruct, we can obtain diverse and correct instruction-code pairs for instruction tuning code LLMs to improve the ability of program synthesis.
- Through executing on original code, we generate test cases in a more effective way. In addition, we offer a new perspective on using them as a measure of instruction difficulty.
- After adding semi-instruct data to self-instruct, the performance is better than only increasing self-instruct data. And the combination breaks out the self-instruct's dilemma and enables performance to grow as data increases.

2. Related Work

2.1. Code LLM

Due to the poor performance of general LLMs on program synthesis, some work (Kocetkov et al.,

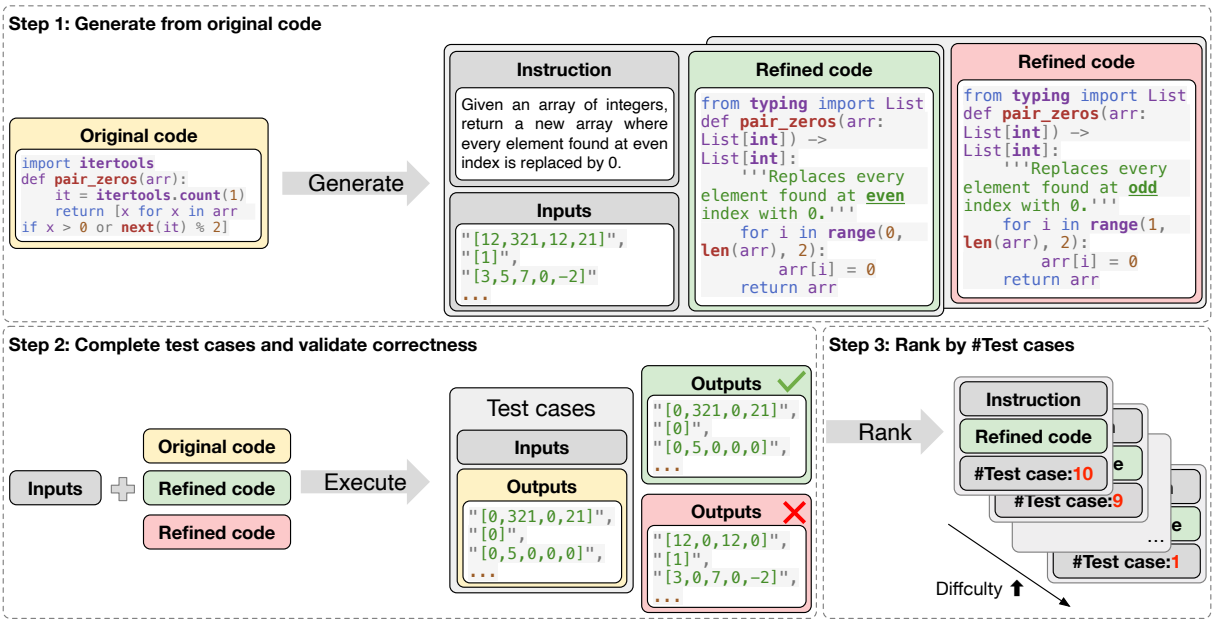

Figure 2: Pipeline of Semi-Instruct. It has three main components. **(1) Generation**: Given the original codes, generate instructions, a fixed number of test cases' inputs based on the instructions, and refined codes; **(2) Validation**: Run the original codes on the inputs, obtain complete test cases through extracting outputs from those successful executions, and subsequently retain refined codes that pass all test cases; **(3) Ranking**: Since the more difficult the instructions are, the less test cases are constructed, sort the data in reverse order according to the number of test cases.

2022; Xu et al., 2022) collect large-scale code data from GitHub, and so on to train specific code LLMs. Pure code LLMs (Li et al., 2022; Fried et al., 2023; Li et al., 2023; Allal et al., 2023) only pre-train on code data from scratch. Others (Chen et al., 2021; Nijkamp et al., 2023; Rozière et al., 2023) use the code data to secondary-pre-train. Although syntax errors appear fewer and fewer in the generated codes (Zhang et al., 2023a), they are limited to generating code based on natural language.

2.2. Code Instruction Tuning

To improve the code LLMs' ability on program synthesis, many works construct aligned instruction-code pairs that match human intentions to finetune the LLMs. The two main types of these data are **natural-instruct** and **self-instruct**.

NI consists of human-written text and code collected from open-source platforms. CodeSearchNet (Husain et al., 2019) collected publicly available GitHub repositories and extracted comments as instructions. Others (Hendrycks et al., 2021; Li et al., 2022; Puri et al., 2021) are derived from online judge websites such as Leetcode, Codeforces, and so on. Problem descriptions are seen as instructions. NI is used by earlier code LLMs (Li et al., 2022; Le et al., 2022; Chen et al., 2021; Wang et al., 2021). Codes in NI are diverse and correct. However, NI suffers from improper code formats and a lack of high-quality instruction-code pairs.

SI is inspired by Alpaca (Taori et al., 2023) and first implemented by Code Alpaca (Chaudhary, 2023). phi-1 (Gunasekar et al., 2023) used GPT-4 to generate a textbook-level high-quality corpus, while WizardCoder (Luo et al., 2023) and Pangu-Coder2 (Shen et al., 2023) used the evol-instruct (Xu et al., 2023) to extend the large data from the basic Code Alpaca. SI has naturally instruction-code paired data and its codes are proper. However, it has low diversity due to generating duplicates and generated codes are uncertain.

2.3. Test Cases

Unlike natural languages, programs can only be considered correct if pass all the test cases written by developers (Hendrycks et al., 2021). Many works proposed to generate test cases automatically to reduce human effort. Earlier heuristic methods based on search(Arcuri, 2017; Lukasczyk and Fraser, 2022) have limitations in diversity and quantity. Later works (Li et al., 2022; Tufano et al., 2021) finetuned pretrained language models on existing labeled data to generate new test cases. Recent works (Chen et al., 2023; Rozière et al., 2023) utilize LLMs to sample without training. However, all of them can't guarantee the correctness of test cases

NI and SI are used separately in all previous work. We combine the two approaches to solve each other's problems. And we propose a novel way to construct test cases effectively and offer a

new perspective on using them.

3. Methodology

Semi-Instruct takes advantage of both natural-instruct and self-instruct. It generates an instruction given an original code to obtain aligned pairs similar to self-instruct, and maintains the diversity and functional correctness of natural-instruct at the same time. Concretely, semi-instruct includes three steps, as shown in Figure 2. First, given an original code, it generates an instruction, a refined code and test case inputs via LLM. Second, verify the diversity of the instruction by its similarity to other instructions, and verify the functional correctness of the refined code on test cases. Note the output of the test case is obtained by executing the input of the test case (generated in the last step) on the original code. Finally, rank the pairs (consist of the instruction and refined code) in descending order based on the difficulty. Here, the difficulty is measured by the number of test cases that the refined code passes. We denote the three steps as generation, validation, and ranking, respectively.

3.1. Generation

The generation phase is not only about converting one-sided data into paired data, but also generate some auxiliary information to verify the quality later.

To collect pairwise data, a corresponding instruction needs to be generated first from the original code. Take inspiration from self-instruct that utilizes LLMs to generate instructions and codes at the same time, the original code can be refined during the generation process to solve the improper code format. This includes expanding nested single-line codes, renaming variables, adding necessary comments, etc. LLMs can generate the refined code with the paired instruction simultaneously, after comprehensively understanding the original code.

Test cases are essential to verify the consistency of the instruction and the correctness of the refined code. The common practice is generating whole test cases consisting of inputs and outputs through LLMs. While most inputs align with the instruction's constraints, LLMs frequently struggle to produce accurate outputs, especially when the instruction describes a complex task that needs logical and numerical reasoning. These inaccurate outputs consume significant generation resources and are challenging to filter out. Therefore, we only generate fixed number of inputs, and introducing a novel approach for test cases construction without generating outputs, described in Sec 3.2.

To use test cases, the answer type needs to be identified, which dictates how the inputs passing into code and how the code return outputs. LLMs can analyze the code and determine its type. If it's call-based, the function name will be extracted. A clear description of each component is below:

Instruction: A clear natural language description used as instruction in the tuning stage. It should directly reflect the function of the original code without too many implement details.

Refined Code: A refined version of the original code used in the tuning stage. It should fix the improper format of the original code, without changing the behavior of the original function.

Answer Type: The way of passing parameters into the original code. It should only be "Call-Based" (receiving input from parameters) or "Standard Input" (reading input from the standard input). If the answer type is "Call-Based", the function name should be included in test cases.

Test Cases: Test cases to validate the refined code based on the requirements of the instruction. Only inputs are generated without outputs during this stage. The existence of the function name is dependent on the answer type.

After adding the original code into the prompt containing definitions of each component with a few examples, we feed the concatenated context into LLMs and extracted each component from the output. The unified prompt template is shown in the appendix. Through the generation stage, the diverse and improper code from NI become proper code with paired instructions. The generations are correspondingly of high diversity. The determined answer type and inputs of test cases will subsequently facilitate the validation phase.

3.2. Validation

Limited by the capabilities of the LLMs, previously generated data needs to be further verified. We propose a novel way to construct completed test cases more effectively and filter out matching, correct, and diverse data in this stage.

First, construct complete test cases by executing the original code on all inputs and gathering the corresponding outputs. Unlike generated outputs, executed outputs are inherently correct due to the correctness of original code. However, some inputs may report runtime errors during execution and result in no outputs. In addition to errors of the inputs themselves, the reason could be a mismatch between instruction and the original code, since the inputs are generated based on instruction. We directly discard the data that all inputs result in no outputs. The remaining data have different numbers of test cases but at least one. Our approach of constructing test cases saves the generation resources and ensures correctness.

Second, check the correctness of the refined code. LLMs naturally cannot ensure the generated code is correct. But the correctness of the original

code can be used to verify the refined codes, by using the complete test cases constructed earlier. If there are inputs that report errors when the refined code is running, or if the results don't match the gold outputs, we assume that the refined code doesn't keep functional consistency with the original code. Only when all the cases pass, the refined code is considered correct and retained.

Last, too similar data are removed. Original codes used to solve the same problem have the potential to generate very similar instructions, which can confuse the model. But this kind of data can also help the model generate more diverse codes. Therefore, we perform a looser filtering based on the ROUGE-L score of the instruction. refined code diversity is inherited from the original code, and the instruction diversity is guaranteed by the filtering.

We use the correctness of the original code to construct test cases and then pick out the matching instruction and the correct refined code. After filtering based on the similarity of instruction, the original code from NI is finally converted to a correct instruction-code pair for instruction tuning.

3.3. Ranking

Test cases are not only used to validate the correctness of refined codes but also can be seen in a new perspective – as a measure of difficulty.

Intuitively, the quality of generated inputs depends on to what extent the LLMs understand the instruction. When the instruction is more complex, the constraint to inputs is more stricter. Although a fixed number of inputs is generated, only those who fulfil the requirements of instruction can extract outputs after executions. So the number of test cases constructed can be seen as a measure of difficulty. Curriculum learning(Bengio et al., 2009) points out that training data should be from easy to hard. We rank the data in reverse order by the number of test cases so that the model can learn incrementally.

The process of semi-instruct is described above. The original code from NI is diverse but improper. First generate corresponding instructions, proper refined code, and test cases' inputs by leveraging the generative capability of LLMs like what SI does. Executing the correct original code on inputs can extract gold outputs from successful results to construct test cases. The complete test cases will exclude incorrect refined code. Semi-instruct bridges NI and SI with both advantages. The construct and usage of test cases are novel and effective.

4. Experiments

NI and SI dataset are constructed and SemI dataset are converted from NI dataset. We present extensive experiments on a widely used LLM and dataset to show how SemI benefits the tuning process.

4.1. Dataset Construction

Natural-Instruct Dataset It is not only as a performance comparison but also used to generate SemI dataset. We choose two common datasets **APPS** (Hendrycks et al., 2021) and **CodeContest** (Li et al., 2022) to serve as foundation. APPS is a collection of 10k coding problems from Codeforces, Leetcod, etc.. The train split has 5k problems with more than 12k Python solutions. CodeContest is a competitive programming dataset scraped from AtCoder, CodeChef, and two existing datasets Description2Code (Caballero and Sutskever, 2016) and CodeNet (Puri et al., 2021). CodeContest has more than 13k problems in train split and solutions are written in several languages like Python, Java, and C++. Correct and incorrect solutions are both contained in the original CodeContest. We only retained the correct Python solutions. For the two datasets, instructions are problem descriptions and codes are solutions.

Before merging into the NI dataset, it is necessary to address the limitations of the two datasets. The following optimisation are implemented. Delete the problems that need special judge. The rest problems' solutions only need to send input from the command line or parameters passing and the outputs can be directly printed. Filter out the solutions whose number of tokens is more than 1k. Long solutions are mostly caused by too many meaningless comments or codes. Merge the solutions from the same problems. One problem may appear on multiple sites and harvest solutions submitted by different users. Limit the number of solutions per problem to a maximum of 25 to make the distribution less sharp. Since several problems have more than 1k solutions in the original two datasets, many have less than 10 solutions.

These approaches can make the NI dataset more realistic and convenient to generate the SemI Dataset. In the end, the natural-instruct dataset has nearly 8k instructions and 126k codes.

Self-Instruct Dataset It is used as the baseline and combined before SemI dataset later. The Code Alpaca project (Chaudhary, 2023) aims to build and share an instruction-following Llama model for code generation. We extend its 20k instruction-code pair data generated by text-davinci-003 to 70k through the same self-instruct techniques.

Semi-Instruct Dataset We use the original codes from NI dataset to generate SemI dataset. 126k codes are sent to LLM and 92k of them generate instruction, refined code, answer type, and test cases' input successfully. After executing the original code

on inputs, nearly 69k piece of data construct at least one test case. The number of refined codes that pass all test cases is 54k. Filtering similar instruction whose ROUGE-L scores with previous data is more than 0.7. Finally, the semi-instruct data have 40k instruction-code pairs.

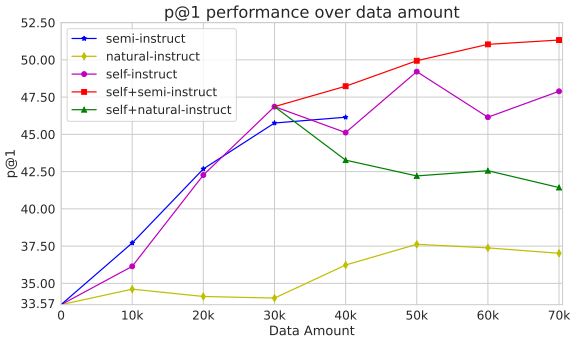

Figure 3: p@1 results on the HumanEval dataset. Note that unlike self-instruct and semi-instruct, the total amount of data for semi-instruct is only 40k.

Data Selection & Order We conducted experiments on datasets ranging from 10k to 70k entries. Each dataset type had its unique selection method. For the NI datasets, we randomize the problems first. Then, we assemble the related solutions. Data are segmented at each data scale. This ensures new problems were introduced with each data increment. Before training, we sort the data randomly. For the SI dataset, data are used in the generated order. The SemI dataset is treated similarly to NI dataset. However, we sort it by the number of test cases before tuning. These methods aim for real-world emulation and bias minimization.

4.2. Experiment Setup

Model The base model we choose for tunning is StarCoder. It is an open-source 15B parameter Code LLM trained on 1T tokens from GitHub and then finetuned on 35B Python tokens. The performance of StarCoder matches OpenAI code-cushman-001 model on HumanEval (Li et al., 2023; Chen et al., 2021). We follow previous work (Luo et al., 2023) to generate data by ChatGPT.

Evaluation Dataset We choose the widely-used dataset – **HumanEval**. It consists of 164 Python programming problems used to judge the performance of a model's ability on code generation. Each problem is provided a function name with docstring so the model can then generate the code. Dataset has test cases to check the functional correctness of generated code.

Metrics We use the average pass@k score of all problems as the metric of performance evaluation. We set $k \in \{1, 10, 100\}$ and sample 200 times for each problem. p@1 is the proportion of all samples that are correct, which can strictly reflect the correctness. p@10, p@100 can reflect the diversity, the more questions are included in the correct samples the higher these two indicators are.

Dataset	p@1	p@10	p@100
base	46.86%	59.62%	66.09%
+ self-instruct	45.12%	60.23%	66.92%
+ natural-instruct	43.27%	57.81%	63.99%
+ semi-instruct	**48.23%**	**65.10%**	**75.01%**

Table 1: Results among adding 10k self-instruct/natural-instruct/semi-instruct data after 30k self-instruct data. "base" represents the performance of base 30k self-instruct data. The best results are marked with **bold**.

4.3. Implementation Details

We use the same hyper-parameters from previous work (Luo et al., 2023) such as limiting the train epochs to 3, the learning rate to 2e-5, the maximum data length to 2048, and the warm-up steps to 30. During the inference phase, we set the temperature to 0.2 and top_p to 0.95 as common settings in previous work to balance the randomness and determinism. We sample 200 times for each problem and calculate pass@k as the metric.

4.4. Results

To fully compare quality and characteristics of NI, SI, and SemI datasets, we conduct five distinct sets of experiments. The first set exclusively utilizes one of the NI, SI, and SemI datasets. Subsequent experiments merge the SI dataset with the SemI or the NI dataset. Mixed datasets are directly combined, maintaining each original order in their respective datasets. A visual representation detailing the p@1 metrics can be found in Fig 3.

Single Type Data The results show that NI only contributes to little performance enhancements, and such gains are often inconsistent. One plausible explanation for the initial decline in performance could be the improper nature of the codes and in NI. This ambiguity potentially misguides model generation. However, as the data scale grows, model start to fit. The impact of such irregularities appears to diminish. The performance decline at later stages might be attributed to the number of instructions added being too small.

SI demonstrates a notable uplift in performance. However, this improvement does not consistently manifest across varying data scales. This suggests that self-instructed data representations like proper code style are more likely to be understood and learned by models. Reasons for instability could be the lack of sufficient data diversity. As we progressed, there was an apparent rise in duplicated data. Another reason could be incorrect data. The accumulation could potentially distort the model's comprehension of the instructions, subsequently influencing its code generation capabilities.

SemI consistently demonstrates an encouraging trend of improvement. Although SemI is obtained from NI dataset, its huge advantage in performance over NI shows that our method significantly solves the problems of NI. It also shows that the intrinsic value of NI is imprisoned by the simple form and is stimulated by SemI. Compared to SI, SemI only slightly underperforms at 30k but achieves superior results on all the rest data scales. It shows that SemI leverages the benefits of the SI through a SI-like approach. The consistent ascent in performance also underscores its robustness.

In synthesizing the outcomes above, it becomes evident that NI and SI have limitations that impact their performance. SemI that bridging the two approaches shows promising results.

Combined Data SemI data is combined with the data of NI and the approach of SI, which shows potential results. We are curious about the upper limit after further combining the data of SI with SemI data. The performance of SI increases steadily from 10k to 30k, but a large drop occurs at 40k. Therefore, when combining the data, we start with 30k self-instruct and follow it with SemI or NI.

We observe that combining SI with NI resulted in a decline in performance. The distinct distributions of these two datasets likely cause this outcome. Merely merging them seems to merge the individual shortcomings. As detailed in Table 1, compared to adding SI, adding NI doesn't augment diversity metrics like p@10 and p@100. In contrast, it reduces them. However, by converting NI dataset into SemI dataset, the performance keeps improving and significantly outperforms SI. Firstly, because SemI dataset is also generated by LLM like SI dataset, the code is more proper and the instruction is closer to the model's expression, the model can learn more efficiently and directly. When only 10k new data are added, compared to SI, SemI improves on p@1 by 3.11%. In all scales, we outperform SI with an average improvement of more than 3%. Second, SemI inherits the diversity of codes in NI. New added data is not duplicated with SI. We improved over SI on p@10 by 4.87% and on p@100 by 8.09% in Table 1. This shows the great advantage of our method in terms of diversity. Finally, SemI has strong robustness, offering consistent performance enhancements.

5. Discussion

5.1. Ablation Study

To have a deep understanding of the function and importance of each component of SemI, we conduct ablation experiments. To strengthen the reliability, we do a total of 40k data and a total of 50k data respectively, and the main results are shown in Table 2. Reducing any of the components in each data scale causes a serious drop in p@1. This means that every component is necessary.

Instructions Replacing instructions with problem descriptions in NI causes performance degradation at both data scales, even lower than SI without adding new data. When the data scale increases, p@1 drops even more. This is because the increment of problem descriptions is not much compared to a large increase in the number of codes. Instead, more data such as this slows down the model's previous ability to understand and generate.

Refined code Replacing refined codes with original codes causes a serious degradation at 40k of data, but recovers a little at 50k. This is because The original codes from NI are improper, and different greatly with SI's codes, misleading the model. However, as more such data is added, the model can slowly understand it. This demonstrates the strong adaptive ability of models. It is crucial to note that boosting this capability comes at the expense of previous performance. Adding 20k of such data remains less optimal than without it.

Both When replacing both instructions and refined codes at the same time, we have empirically found that this is better than replacing one alone. This may stem from the internal consistency of NI data - both problem descriptions and original codes are written by people. Comparison to "+semi-instruct" can also be a good way to improve the diversity of SemI, where direct combining of NI and SI does not transfer its diversity, but rather reduces it at p@10, p@100.

Sort Two main orders that are considered in terms of difficulty exist in experiments. One is that the SemI data is in reverse order by the number of test cases, where the harder the problem the fewer test cases will be left; the other is that the SemI dataset is combined after the SI dataset, with the SI considered to be simpler than SemI. Removing the test cases sort not only regresses performance,

Dataset	add 10k data			add 20k data		
	p@1	p@10	p@100	p@1	p@10	p@100
base	45.12%	60.23%	66.92%	45.12%	60.23%	66.92%
+ semi-instruct	**48.23%**	**65.10%**	**75.01%**	**49.94%**	**67.90%**	**75.37%**
- instructions	40.43%	52.65%	61.00%	39.72%	55.63%	68.06%
- refined code	38.38%	54.95%	66.46%	41.12%	55.39%	63.47%
- both	43.27%	57.81%	63.99%	42.21%	56.43%	65.71%
- test cases sort	44.39%	62.33%	71.88%	44.74%	64.53%	73.56%
- all sort	46.15%	65.74%	76.16%	44.46%	62.60%	72.46%

Table 2: Ablation study on semi-instruct. The left side “add 10k data" means add 10k new data to 30k self-instruct data, while the right side is adding 20k new data. “base” represents the performance of base 30k self-instruct data. “+semi-instruct" means add semi-instruct data to “self-instruct". “-instructions" means replace instructions in added semi-instruct data with their problem descriptions in natural-instruct dataset. “-refined code" means replace refined codes with original codes. “-both" means replace both. “-test cases sort" means only removing ranking in semi-instruct. “-all sort" means random shuffle all 40k/50k data include self-instruct and semi-instruct. The best results are marked with **bold**.

but the degree of model improvement grows much more slowly when the data scale increases. This suggests that ranking can be effective in improving the efficiency of model learning. When adding 10k SemI data, although “- all sort” will be lower on p@1 by 2.08%, there is an increase on p@10, p@100. This suggests that disrupting the data can slightly increase the diversity of model generation. But when the amount of data increases, this advantage disappears and regression occurs on all metrics. This proves our hypothesis, as the data in NI comes from the competition websites, and SemI inherits significantly higher difficulty than SI.

Through sufficient experiments, we validated each step of the SemI. We avoided the problems of missing instructions and improper codes in NI through LLM, and distinguished the data difficulty through test cases. The performance improvement proved the rationality and superiority of our method.

5.2. Case Study

In addition to the validation of correctness and diversity, we also perform a direct analysis of the codes generated by the models with different data finetune. We compare the generating codes from model training by NI dataset and SemI dataset, as shown in Figure 4 respectively. NI generates only one nested line of code that fails the test, while SemI generates a clear and correct code.

For some high-level programmers, it is a common trick to write multiple lines of code as one nested line. Such improper codes are more complex and difficult to implement than multi-line codes. NI has a large amount of such code, which adds unnecessary complexity and difficulty. The model mimics human behavior by generating nested single-line code but fails to deal with the logic clearly, resulting in the generation of incorrect code.

From a problem-solving perspective, there is no fundamental difference between nested code and proper code. Compared to NI, SemI's code is more standardized, with docstring introducing the whole function, and each key step is annotated, each line is not nested, which not only increases readability but also avoids complex logic. Through SemI, a large number of improper codes from NI are converted into proper codes, which greatly reduces the difficulty to learn and improves the accuracy.

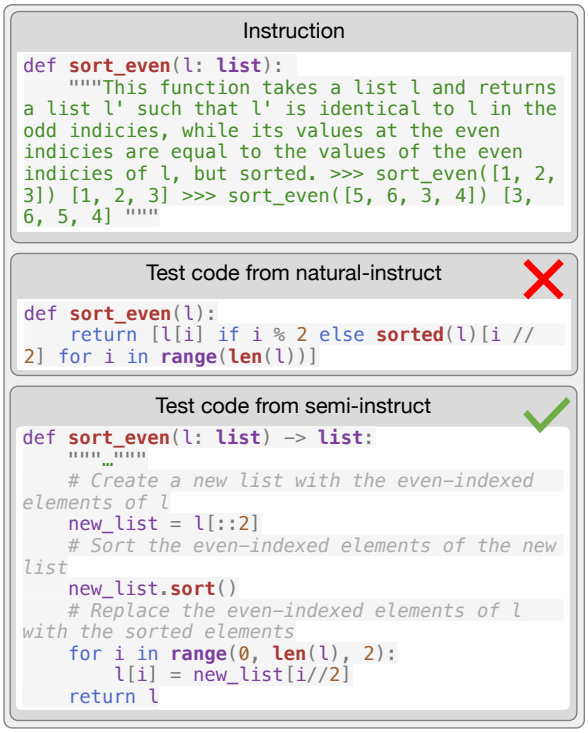

Figure 4: Code generated for the same problem after training the model using the natural-instruct dataset and the semi-instruct dataset, respectively. The former is wrong, and the latter is correct.

6. Conclusion

In this paper, we propose a novel way of collecting instruction tunning data for code LLMs, semi-instruct, which combines NI and SI. We take the diverse but improper codes in NI and generate proper codes with aligned instructions through a SI-like generation process. Generated codes can't be guaranteed correct. To cope with it, we generate test cases' inputs and execute correct codes in NI to get outputs. The complete test cases can be used to test the correctness of the generated codes. The experiments demonstrate that our method effectively combines the strength of NI and SI.

7. Bibliographical References

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

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. Santacoder: don't reach for the stars!

Andrea Arcuri. 2017. Many independent objective (mio) algorithm for test suite generation. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*, pages 3–17. Springer.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA. Association for Computing Machinery.

Johannes Bohnet and Jürgen Döllner. 2011. Monitoring code quality and development activity by software maps. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, page 9–16, New York, NY, USA. Association for Computing Machinery.

Ethan Caballero and Ilya Sutskever. 2016. Description2code dataset.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 21314–21328. Curran Associates, Inc.

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*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172.

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

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2:90–121.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2022. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, Dublin, Ireland. Association for Computational Linguistics.

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. *arXiv preprint arXiv:2308.07124*.

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.

Baptiste Rozière, 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*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback.

Richard Shin and Benjamin Van Durme. 2022. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5417–5425, Seattle, United States. Association for Computational Linguistics.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023a. 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, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

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

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. *arXiv preprint arXiv:2304.12244*.

Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating natural language proofs with verifier-guided search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 89–105, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. 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 769–787. Association for Computational Linguistics.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023b. Siren's song in the ai ocean: A survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*.

8. Language Resource References

Hendrycks, Dan and Basart, Steven and Kadavath, Saurav and Mazeika, Mantas and Arora, Akul and Guo, Ethan and Burns, Collin and Puranik, Samir and He, Horace and Song, Dawn and Steinhardt, Jacob. 2021. *Measuring Coding Challenge Competence With APPS*. Curran. PID https://github.com/hendrycks/apps.

Husain, Hamel and Wu, Ho-Hsiang and Gazit, Tiferet and Allamanis, Miltiadis and Brockschmidt, Marc. 2019. *Codesearchnet challenge: Evaluating the state of semantic code search*. PID https://huggingface.co/datasets/code_search_net.

Kocetkov, Denis and Li, Raymond and Allal, Loubna Ben and Li, Jia and Mou, Chenghao and Ferrandis, Carlos Muñoz and Jernite, Yacine and Mitchell, Margaret and Hughes, Sean and Wolf, Thomas and others. 2022. *The stack: 3 tb of permissively licensed source code*. PID https://huggingface.co/datasets/bigcode/the-stack.

Puri, Ruchir and Kung, David and Janssen, Geert and Zhang, Wei and Domeniconi, Giacomo and Zolotov, Vladimir and Dolby, Julian T and Chen, Jie and Choudhury, Mihir and Decker, Lindsey and Thost, Veronika and Thost, Veronika and Buratti, Luca and Pujar, Saurabh and Ramji, Shyam and Finkler, Ulrich and Malaika, Susan and Reiss, Frederick. 2021. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. Curran. PID https://github.com/IBM/Project_CodeNet/tree/main.

Frank F. Xu and Uri Alon and Graham Neubig and Vincent Josua Hellendoorn. 2022. *A systematic evaluation of large language models of code*. ACM. PID https://github.com/VHellendoorn/Code-LMs.