VERSE: Verification-based Self-Play for Code Instructions

Hao Jiang[1], **Qi Liu**[1, 2*], **Rui Li**[1], **Yuze Zhao**[1], **Yixiao Ma**[1],
Shengyu Ye[1], **Junyu Lu**[1, 2], **Yu Su**[2, 3]

[1]State Key Laboratory of Cognitive Intelligence, University of Science and Technology of China
[2]Institute of Artificial Intelligence, Hefei Comprehensive National Science Center
[3]School of Computer Science and Artificial Intelligence, Hefei Normal University
{jianghao0728, ruili2000, yuzezhao, mayx, ysy007, lujunyu}@mail.ustc.edu.cn, {qiliuql}@ustc.edu.cn, yusu@hfnu.edu.cn

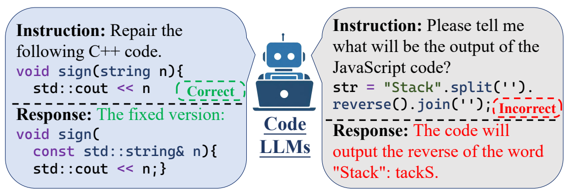
Abstract

Instruction-tuned Code Large Language Models (Code LLMs) have excelled in diverse code-related tasks, such as program synthesis, automated program repair, and code explanation. To collect training datasets for instruction-tuning, a popular method involves having models autonomously generate instructions and corresponding responses. However, the direct generation of responses does not ensure functional correctness, a crucial requirement for generating responses to code instructions. To overcome this, we present Verification-Based Self-Play (VERSE), aiming to enhance model proficiency in generating correct responses. VERSE establishes a robust verification framework that covers various code instructions. Employing VERSE, Code LLMs engage in self-play to generate instructions and corresponding verifications. They evaluate execution results and self-consistency as verification outcomes, using them as scores to rank generated data for self-training. Experiments show that VERSE improves multiple base Code LLMs (average 7.6%) across various languages and tasks on many benchmarks, affirming its effectiveness.

Code — https://github.com/TechxGenus/VERSE

1 Introduction

In the realm of Large Language Models (LLMs), code serves as a pivotal link between human understanding and machine execution. It acts as a fundamental element by transforming human instructions into executable actions, thereby constructing agents (Hong et al. 2023; Li et al. 2024a; Liu et al. 2024). To augment the code generation capabilities of LLMs, Code LLMs, trained on a vast corpus of code, have emerged as a focal point (Li et al. 2022; Johnson, Tarlow, and Walder 2023; Khakhar, Mell, and Bastani 2023; Li et al. 2024b). Recently, researchers are primarily focusing on instruction-tuning (Chung et al. 2022; Ouyang et al. 2022; Longpre et al. 2023) for Code LLMs (Wang et al. 2023b; Muennighoff et al. 2023; Luo et al. 2023). This technique utilizes code-related instructions and their corresponding responses in supervised training, enhancing the model's understanding and generalization abilities across various code downstream tasks, such as program synthesis (Young, Bastani, and Naik 2019), and code explanation (Mahbub, Shuvo, and Rahman 2022).



Instruction: Repair the following C++ code.
```
void sign(string n){
  std::cout << n
```
Response: The fixed version:
```
void sign(
  const std::string& n){
  std::cout << n;}
```

Instruction: Please tell me what will be the output of the JavaScript code?
```
str = "Stack".split('').
reverse().join('');
```
Response: The code will output the reverse of the word "Stack": tackS.

Figure 1: Instructions and responses created by Code LLMs.

To collect training data for instruction-tuning, considering factors such as the cost and copyright issues of using data written by humans, a prevalent approach is to employ models to autonomously generate instructions and responses (Wang et al. 2023a; Taori et al. 2023). However, code instructions demanding strict adherence to the functional correctness of responses—a facet often overlooked in previous research. As illustrated in Figure 1, unverified and unexecuted data generated by LLMs lacks guaranteed quality and may contain errors. A natural idea to solve it is to have Code LLMs verify the functional correctness of self-generated responses with the executable nature of code. There have been some attempts at this in Python code generation tasks (Haluptzok, Bowers, and Kalai 2023; Rozière et al. 2023), but it has not been explored for general code instructions. In this paper, we focus on **how Code LLMs can self-improve through verification for code instructions**.

However, direct verification of responses to code instructions presents challenges. **On the one hand**, responses to code instructions have different modals. For example, generating code summaries or predicting outputs for programs differs from program synthesis and repair, as they may not produce executable code, but generate natural language text that cannot be executed directly instead. This introduces new difficulties to self-verification. **On the other hand**, both the responses and verification schemes are not guaranteed to be entirely correct. Some responses that are deemed correct under one set of verification schemes may be considered incorrect under another set of schemes. Reaching a consensus on the optimal response is not easy.

To address these challenges, in this paper, we present VERSE, an innovative approach that leverages verification-based self-play with execution to improve responses to code instructions for Code LLMs. We initiate the process by creat-

*Corresponding Author.

ing instructions and marking relevant attributes. Code LLMs then transform these instructions into verifications through self-play. Subsequently, Code LLMs generate and combine multiple responses and scripts for execution. The construction of verifications and the combination methods of execution are specifically designed for different modalities of responses, addressing the first challenge. Each response's quality is evaluated based on verification scores derived from execution results and consistency. The responses are then ranked to reach the optimal consensus and added to the candidate data pool. Ultimately, Code LLMs sample training instances from the candidate data pool based on their verification scores to construct datasets for their self-improvement.

To assess the efficacy of VERSE, we conduct experiments across various tasks, encompassing clone detection (Svajlenko et al. 2014), defect detection (Zhou et al. 2019), program synthesis, automated program repair (Tang et al. 2021; Zhao et al. 2024), and code explanation. VERSE exhibits noteworthy enhancements across these diverse tasks. For instance, when applied to different base Code LLMs, metrics for these tasks increase by an average of 7.6%. Our extensive analysis underscores the superiority and significance of VERSE.

2 Preliminaries

In this section, we provide detailed definitions for instructions and code instructions along with their construction to cover related tasks. These definitions serve to formalize the process of VERSE. The related work section can be found in Appendix A.

Definition 1 (Instruction)**.** Let T be the set of text involved in model input and output, with $N \subseteq T$ representing text exclusively in the natural text modality. We define the tuple $x = (x.n, x.t) \in T$ to represent an instruction, where $x.n \in N$ is a natural language requirement, and $x.t \in T$ denotes an (optional) text being operated on. The function $G : X \rightarrow T$ generates responses for instructions.

Definition 2 (Code Instruction)**.** Let $P \subseteq T$ represent text in program modality. We define $C = \{x | (\exists x.t \wedge x.t \in P) \vee G(x) \in P\}$ as the set of code instructions. For any $x \in C$, the associated text $x.t$ or the response $G(x)$ is in program modality.

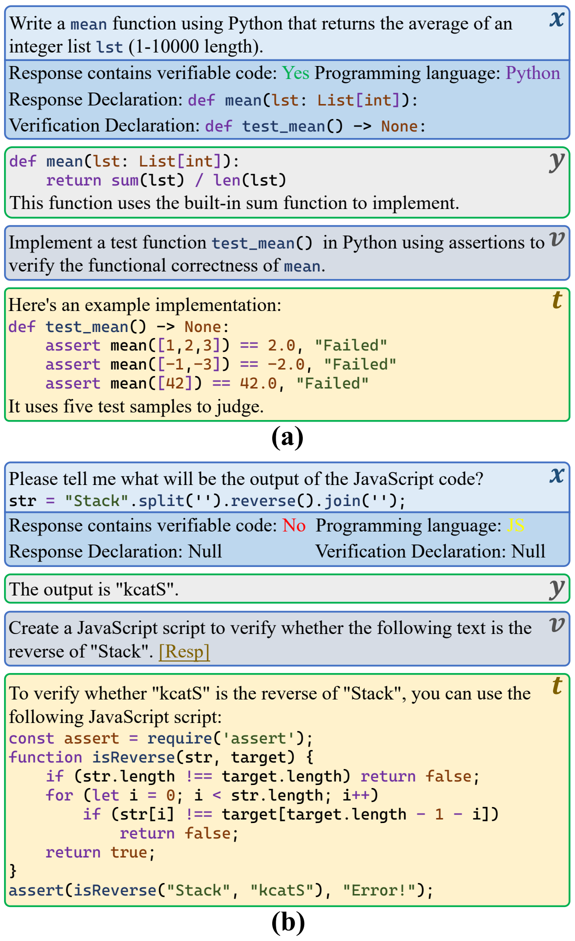
Examples of C are showcased in Figure 1. The natural language parts $x.n$ are *"Repair the C++ . . . "* and *"Please tell me . . . "*, while the code snippets $x.t$ consist of *"void sign . . . "* and *"str = . . . "*.

3 Verification-based Self-Play

In this section, we present a detailed overview of VERSE, elucidating each step of the process. Figure 4 shows the pipeline of VERSE. We use θ to represent the parameters of Code LLMs.

3.1 Automatic Data Generation

Our data generation process consists of three steps: 1) create instructions and responses, 2) create verifications, 3) create scripts. Figures 2 and 3 contain multiple concrete examples.


Write a `mean` function using Python that returns the average of an integer list `lst` (1-10000 length). *x*

Response contains verifiable code: Yes Programming language: Python

Response Declaration: `def mean(lst: List[int]):`

Verification Declaration: `def test_mean() -> None:`

```
def mean(lst: List[int]):
    return sum(lst) / len(lst)
```
This function uses the built-in sum function to implement. *y*

Implement a test function `test_mean()` in Python using assertions to verify the functional correctness of `mean`. *v*

Here's an example implementation: *t*
```
def test_mean() -> None:
    assert mean([1,2,3]) == 2.0, "Failed"
    assert mean([-1,-3]) == -2.0, "Failed"
    assert mean([42]) == 42.0, "Failed"
```
It uses five test samples to judge.

(a)

Please tell me what will be the output of the JavaScript code? *x*
```
str = "Stack".split('').reverse().join('');
```
Response contains verifiable code: No Programming language: JS

Response Declaration: Null Verification Declaration: Null

The output is "kcatS". *y*

Create a JavaScript script to verify whether the following text is the reverse of "Stack". [Resp] *v*

To verify whether "kcatS" is the reverse of "Stack", you can use the following JavaScript script: *t*
```
const assert = require('assert');
function isReverse(str, target) {
    if (str.length !== target.length) return false;
    for (let i = 0; i < str.length; i++)
        if (str[i] !== target[target.length - 1 - i])
            return false;
    return true;
}
assert(isReverse("Stack", "kcatS"), "Error!");
```

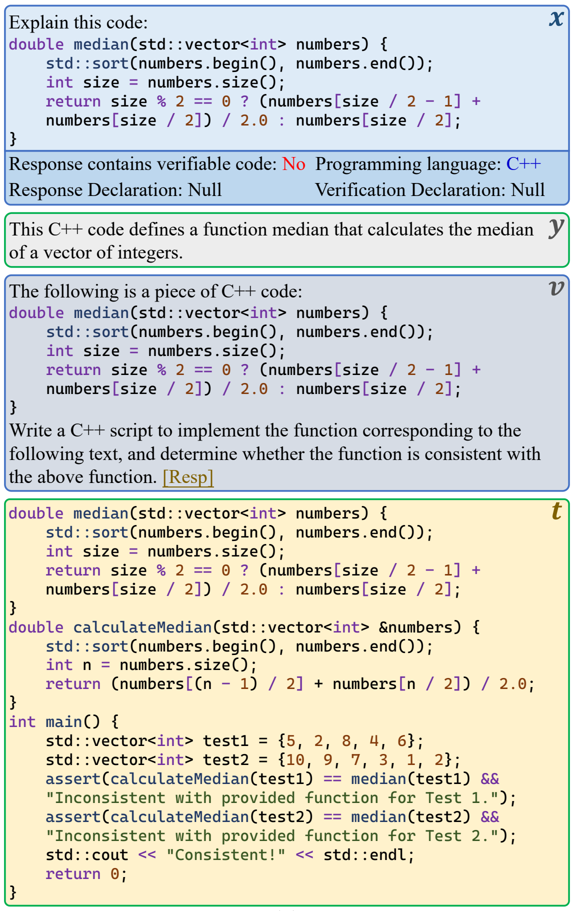
(b)

Figure 2: Some cases of data generated by VERSE, covering a variety of types.

Create Instructions and Responses Utilizing Self-Instruct (Wang et al. 2023a; Taori et al. 2023), our approach generates an initial set of instructions by using diverse seed instructions and leveraging the few-shot capability of LLMs. We mandate the model to mark four additional attributes for each instruction: whether the corresponding response contains verifiable code, programming language, response declaration, and verification declaration. See Figures 2 and 3 for examples of specific attributes. Specifically distinguishing these attributes aids execution in Section 3.2. Finally, we obtain a dataset of instructions.

We use y to represent the response to the instruction x. For each instruction, the model is prompted using few-shot learning to generate multiple responses. As shown in Figure 4, for instruction x, we generate different responses, represented as y_1, y_2 and y_3.

Create Verifications In the subsequent steps, we transform the instructions into verifications - a special type of instruction used to verify the functional correctness of responses. Verifications are designed to prompt the model to generate scripts for assessing the functional correctness of responses.



Explain this code: *x*
```
double median(std::vector<int> numbers) {
    std::sort(numbers.begin(), numbers.end());
    int size = numbers.size();
    return size % 2 == 0 ? (numbers[size / 2 - 1] +
    numbers[size / 2]) / 2.0 : numbers[size / 2];
}
```
Response contains verifiable code: No Programming language: C++
Response Declaration: Null Verification Declaration: Null

This C++ code defines a function median that calculates the median of a vector of integers. *y*

The following is a piece of C++ code: *v*
```
double median(std::vector<int> numbers) {
    std::sort(numbers.begin(), numbers.end());
    int size = numbers.size();
    return size % 2 == 0 ? (numbers[size / 2 - 1] +
    numbers[size / 2]) / 2.0 : numbers[size / 2];
}
```
Write a C++ script to implement the function corresponding to the following text, and determine whether the function is consistent with the above function. [Resp]

t
```
double median(std::vector<int> numbers) {
    std::sort(numbers.begin(), numbers.end());
    int size = numbers.size();
    return size % 2 == 0 ? (numbers[size / 2 - 1] +
    numbers[size / 2]) / 2.0 : numbers[size / 2];
}
double calculateMedian(std::vector<int> &numbers) {
    std::sort(numbers.begin(), numbers.end());
    int n = numbers.size();
    return (numbers[(n - 1) / 2] + numbers[n / 2]) / 2.0;
}
int main() {
    std::vector<int> test1 = {5, 2, 8, 4, 6};
    std::vector<int> test2 = {10, 9, 7, 3, 1, 2};
    assert(calculateMedian(test1) == median(test1) &&
    "Inconsistent with provided function for Test 1.");
    assert(calculateMedian(test2) == median(test2) &&
    "Inconsistent with provided function for Test 2.");
    std::cout << "Consistent!" << std::endl;
    return 0;
}
```

(c)

Figure 3: Some cases of data generated by VERSE, covering a variety of types.

For an instruction x, we use few-shot learning to enable the model to generate the corresponding verification v. We collect multiple verifications per instruction to account for diversity and multi perspectives in verification approaches. For example, a program synthesis instruction can be verified either through static formal verification or by running randomly generated test samples.

During this step, the presence of code in the response y is crucial. If $y \in P$, like verifying a function as shown in case (a) of Figure 2, we prompt the model to generate test scripts, unrelated to the specific function implementation. If $y \notin P$, which means the response y is in natural language, like cases (b) and (c) in Figures 2 and 3, direct execution is impossible. In such cases, verification is response-specific. We prompt Code LLMs to generate verification ideas and combine them with responses. In case (a), for instance, the verification idea is to generate a program checking if the response is a string reversal; then, the virtual text link "[Resp]" is replaced with the content of response y, to generate the actual prompt for generating test scripts. This enables verifying different responses using the same idea, ensuring a fair evaluation of functional correctness for each response.

Create Scripts As verification is a special kind of instruction, similar to the step for generating responses, we enable the model to generate the test script t for the verification v. Regardless of whether the response y contains code, the script t contains an executable program for testing.

3.2 Post-processing and Execution

We extract code snippets from response y and script t, aggregating them for execution. The code snippet extracted from response y is denoted as $\overline{y}$, and from script t is denoted as $\overline{t}$. If the response is in program modality, denoted as $y \in P$, we combine $\overline{y}$ and $\overline{t}$ for execution. If $y \notin P$, we extract only $\overline{t}$ for execution.

The process of aggregating programs considers verifiability and programming language, selecting the execution entry point based on declarations mentioned in Section 3.1. If there is no relevant declaration, the entire extracted code is executed. The program runs in a multi-language sandbox with common dependencies for security, yielding final execution results. We denote the result using the function $\mathrm{r}(\overline{y}, \overline{t})$, which evaluates to 1 for successful execution and 0 in case of an error, as shown in Figure 4.

3.3 Consistency-driven Data Ranking

In this subsection, we outline the process for ranking the quality of responses with execution results and self-consistency. Inspired by intra-consistency and inter-consistency (Jung et al. 2022; Huang et al. 2023), we define two types of consistency based on the equivalence of responses and the results of execution, combined with the model's confidence in the verification, to calculate the verification scores for data ranking. We provide the definitions for both types of consistency first, and then we introduce the calculation of verification scores. Examples of both types of consistency are provided in Figure 4.

Equivalence-based Consistency If multiple responses share the same functionality, they are more likely to be correct. Therefore, for a specified function, we calculate the probability of the model generating a response equivalent to it for data ranking. We measure the ratio of responses implementing the desired function to quantify equivalence. To address difficulties in establishing direct equivalence, we draw inspiration from dynamic equivalence (Gulwani, Radicek, and Zuleger 2016; Hu et al. 2019) and self-consistency (Wang et al. 2022). We define two responses as equivalent if they consistently exhibit the same behavior across an extensive set of verifications.

We organize responses by grouping them with identical functions based on their performance in self-generated verifications. We denote the group corresponding to y as c^y. As shown in Figure 4, y_1 belongs to c^{y_1}, and y_2 and y_3 both belong to c^{y_2}. Subsequently, we compute the probability of generating a response with the required functionality c:

$$\mathcal{P}(c|x) = \mathbb{E}_y[y \in c] \tag{1}$$

Execution-based Consistency We also measure consistency between a response and its corresponding test scripts.

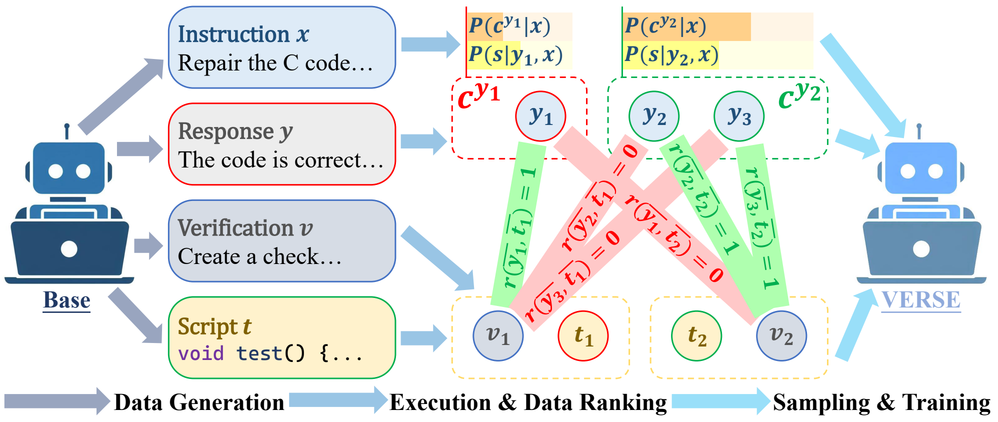


Figure 4: Pipeline of VERSE, illustrating the key steps.

We calculate the probability of successful execution of the post-processed program, denoted as $\mathcal{P}(s|y,x)$, using the following formula:

$$\mathcal{P}(s|y,x) = \mathbb{E}_v[\mathcal{P}(s|v,y,x)] = \mathbb{E}_{v,t}[\mathrm{r}(\overline{y},\overline{t})] \quad (2)$$

Verification score for Data Ranking We derive the verification score $\mathcal{R}(y|x)$ for each response by systematically integrating consistency factors, which act as criteria for data ranking. Our basic method involves calculating the product of these factors, as described below:

$$\mathcal{R}(y|x) = \mathcal{P}(c^y|x)\mathcal{P}(s|y,x) \quad (3)$$

However, verification difficulty varies among code instructions. For example, repair instructions with error reporting functionality are relatively simple, while verifying answers to competition-level problems is more difficult. Considering this variability, we assign weights to relevant probabilities. To assess the model's confidence in the verification scheme, we employ two metrics: entropy calculation for verifications, and the evaluation of consistency among verification schemes across various responses. The respective formulas for these metrics are as follows:

$$\mathcal{H}(v|x) = -\mathbb{E}_v[\log \mathcal{P}(v|x)], \mathcal{P}(s|x) = \mathbb{E}_y[\mathcal{P}(s|x,y)] \quad (4)$$

We evaluate verification confidence using the product of $\mathcal{P}(s|x)$ and the reciprocal of $\mathcal{H}(v|x)$ as weights for respective probabilities. This method is model and instruction agnostic. When the response passes inconsistent verification schemes or the entropy of generated verifications is high, verification confidence is low, and priority is given to referencing the equivalence-based consistency score. Conversely, a higher weight is assigned to the execution-based consistency score. The calculation formula is as follows:

$$w = \alpha\mathcal{P}(s|x)\mathcal{H}(v|x)^{-1}, \mathcal{R}(y|x) = \mathcal{P}(c^y|x)\mathcal{P}(s|y,x)^w \quad (5)$$

Here, α serves as a hyperparameter, representing a prior on the importance of two types of consistency. The analysis of this design and hyperparameter is detailed in Appendix D.1. We rank generated responses by verification scores and integrate them into the candidate data pool.

3.4 Sampling and Training

We use rejection sampling fine-tuning (Dong et al. 2023) to train the model. We select the responses with the highest verification scores corresponding to each instruction data from the candidate data pool and combine them to train the model. In cases of consistent verification scores from multiple samples, we randomly select samples for training.

4 Experimental Setup

4.1 Code LLMs

In our experiments, we use two popular open-source Code LLMs: CodeLlama (Rozière et al. 2023), pre-trained on an extensive code dataset, and Deepseek-Coder (Guo et al. 2024), trained from scratch. We mainly utilize CodeLlama-7B and Deepseek-Coder-6.7B-Base to conduct primary experiments; models with other sizes are used for comparison and analysis.

4.2 Initialization

For LLMs, the training process usually adopts multiple stages, and the next stage of training starts from checkpoints of the previous stage (Ouyang et al. 2022). In our experiments, we start from both base Code LLMs and checkpoints after fine-tuning with some data to fully prove the effectiveness of VERSE.

To start training using base Code LLMs, we employ the original Self-Instruct pipeline as a baseline to generate an equivalent amount of training data without the verification step. For training with tuned checkpoints, we use two publicly popular datasets: CodeAlpaca-20k and Evol-instruction-66k to build the initial checkpoints. We also compare them with the evaluation results of the official instruction versions for both models. Additional discussions can be found in Appendix E.

Task	Category	Dataset	Language	Size	Metric
CD	N + P → N	BigCloneBench	Java	2,000	F1
DD	N + P → N	Devign	C	2,000	Acc
PS	N → P	HumanEvalSynthesis	Python C++ Java JS Go Rust	164	Pass@1
APR	N + P → P	HumanEvalFix	Python C++ Java JS Go Rust	164	Pass@1
CE	N + P → N	HumanEvalExplain	Python C++ Java JS Go Rust	164	Pass@1

Table 1: Summary of benchmarks and datasets used in this paper.

4.3 Data Generation

We utilize vLLM (Kwon et al. 2023) for accelerated sampling in generation. We consider data repeatability and generation quality (see Appendix B.1 for details). In our experiment, for both CodeLlama-7B and Deepseek-Coder-6.7B-Base, we collect a dataset comprising 100,000 instructions. Each instruction is associated with 20 responses and 20 verification-script pairs. The impact of sampling quantity and quantitative analysis are discussed in Appendix D.2 and Appendix D.3.

4.4 Execution Details

We execute programs within a sandbox that supports multiple programming languages. To ensure execution security, we limit the use of associated hardware resources. Details of the execution environment are available in Appendix B.2. We compute the verification score for each response. If the verification score for an instruction is 0, the instruction is deemed invalid. Ultimately, for different models, we obtain around 50,000 valid instructions with corresponding responses and verification scores. To ensure a fair comparison, we retain 40,000 valid instructions for training, adjusting for variations in the amount of valid data obtained by different models.

4.5 Training Settings

Our models are trained on 2 Nvidia A100 GPUs for 2 epochs using the Transformers library. We employ Alpaca-style instruction templates (Taori et al. 2023) for training, and we set the hyperparameter α for calculating the verification score to 4. Memory efficiency and speed are enhanced through techniques, including Deepspeed ZeRO3 (Rajbhandari et al. 2019) and FlashAttention2 (Dao 2023). We configure a batch size per GPU of 32, a maximum sequence length of 2048, and a learning rate of 5e-5. Training employs the Adafactor optimizer (Shazeer and Stern 2018), coupled with a cosine scheduler featuring 15 warm-up steps.

5 Evaluation

We introduce the benchmarks used in the experiments in Section 5.1. We report the main results of the experiments in Section 5.2 and conduct some analysis in Section 5.3. In the following sections, Section 5.4, Section 5.5, and Section 5.6, we perform additional experiments and analysis.

5.1 Benchmarks

Our experiments encompass five tasks from two widely recognized benchmarks, CODEXGLUE (Lu et al. 2021) and HUMANEVALPACK (Muennighoff et al. 2023), these tasks aim to evaluate the quality of responses to diverse code instructions. The first two tasks are collected from the CODEXGLUE benchmark, while the last three tasks are collected from the HUMANEVALPACK benchmark. The specific tasks are as follows:

- **Clone Detection (CD):** Determine whether two pieces of code belong to the same function.
- **Defect Detection (DD):** Identify whether a code snippet contains defects.
- **Program Synthesis (PS):** Synthesize programs to specified functional requirements.
- **Automated Program Repair (APR):** Fix error functions that fail during execution.
- **Code Explanation (CE):** Generate a relevant functional description for a given function.

Three tasks in HUMANEVALPACK are extended from the HUMANEVAL dataset (Cassano et al. 2023). We use them to examine models responding to various code instructions for evaluating the effectiveness of VERSE. Please see Table 1 and Appendix C for details of the evaluation metrics and datasets for various tasks.

5.2 Primary Results

We report the evaluation results of CodeLlama-7B and Deepseek-Coder-6.7B-Base after training with VERSE. The results and comparisons of training from different models are shown in Table 2, Table 3, Figure 5. The results shown for tasks in HUMANEVALPACK are average results for each language. Detailed results for each programming language and task can be found in Appendix F.

The experimental results highlight VERSE's outstanding performance in enhancing Code LLMs for generating correct responses to code instructions. Base Code LLMs trained with VERSE show a significant average improvement of 7.6% across various tasks. Models fine-tuned on CodeAlpaca-20k (20 examples in total) and Evol-instruction-66k (66 high-quality examples in total) checkpoints demonstrate an average improvement of 7.6% and 3.5%. Additionally, the best model trained with VERSE outperforms the official instruction versions of both models on multiple tasks and average metrics, performing well at limited training scale.

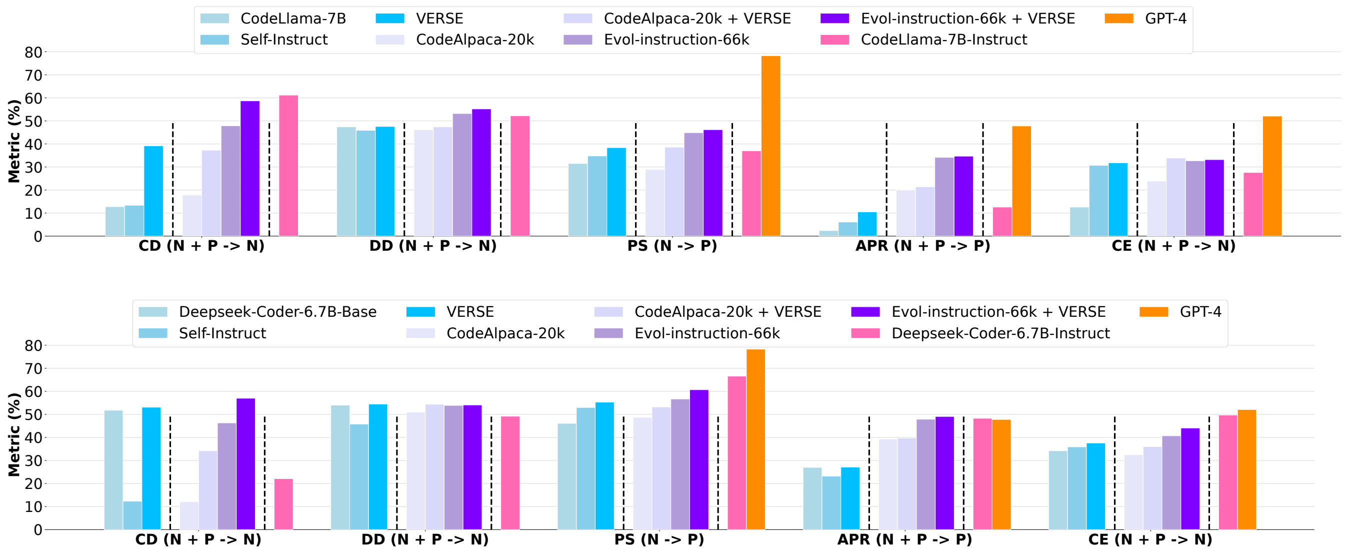


Figure 5: Various comparison results of models trained based on CodeLlama-7B and Deepseek-Coder-6.7B-Base. (**Blue**) Models trained starting from the base model. (**Purple**) Models trained starting from fine-tuned checkpoints. (**Red**) CodeLlama-7B-Instruct, Deepseek-Coder-6.7B-Instruct and GPT-4.

Method	**CD**	**DD**	**PS**	**APR**	**CE**	**Avg.**
From CL-7B:						
Base	12.8	47.4	31.5	2.4	12.6	21.3
Self-Instruct	13.4	45.9	34.8	6.1	30.7	26.2
VERSE	**39.2**	**47.6**	**38.4**	**10.5**	**31.8**	**33.5**
From checkpoints fine-tuned using CodeAlpaca-20k:						
w/o VERSE	17.8	46.2	29.0	20.0	23.9	27.4
VERSE	**37.3**	**47.4**	**38.6**	**21.4**	**33.9**	**35.7**
From checkpoints fine-tuned using Evol-instruction-66k:						
w/o VERSE	47.9	53.2	44.9	34.2	32.7	42.6
VERSE	**58.7**	**55.2**	**46.2**	**34.7**	**33.2**	**45.6**
CL-7B-Instruct	61.2	52.2	37.0	12.6	27.6	38.1

Table 2: The evaluation results of training from CodeLlama-7B (CL-7B) using VERSE.

Method	**CD**	**DD**	**PS**	**APR**	**CE**	**Avg.**
From DS-6.7B-Base:						
Base	51.8	54.0	46.1	27.0	34.2	42.6
Self-Instruct	12.3	45.8	53.0	23.2	35.9	34.0
VERSE	**53.1**	**54.5**	**55.3**	**27.1**	**37.6**	**45.5**
From checkpoints fine-tuned using CodeAlpaca-20k:						
w/o VERSE	12.1	51.0	48.7	39.3	32.5	36.7
VERSE	**34.2**	**54.4**	**53.2**	**39.7**	**36.0**	**43.5**
From checkpoints fine-tuned using Evol-instruction-66k:						
w/o VERSE	46.3	53.9	56.7	47.9	40.7	49.1
VERSE	**57.0**	**54.1**	**60.7**	**49.1**	**44.1**	**53.0**
DS-6.7B-Instruct	22.1	49.2	66.6	48.3	49.7	47.2

Table 3: The evaluation results of training from Deepseek-Coder-6.7B-Base (DS-6.7B) using VERSE.

5.3 Analysis

We further analyze the specific results in various tasks and find that VERSE helps different tasks differently. VERSE ensures the quality of data for some tasks like clone detection, program synthesis, and code explanation when the model directly responds without execution. When using Self-Instruct or fine-tuning with CodeAlpaca-20k, we are surprised to find that their metrics on some tasks even drop compared to the base model. The official instruction version, Deepseek-Coder-6.7B-Instruct, also exhibits a similar phenomenon on the clone detection and defect detection task. This further confirms the importance of ensuring data quality. Data generated without verification may contain errors that can greatly damage the model.

Evaluation results demonstrate VERSE's significant improvements in clone detection, program synthesis, and code explanation tasks. However, its impact on defect detection and automated program repair tasks is limited. Our analysis suggests that the primary reason lies in the model's difficulty in generating code resembling genuine human errors. Even GPT-4, when tasked with producing code with syntax errors, often fails to comply. High-quality training data for LLMs actually hinders the creation of incorrect inputs, aligning with previous studies using synthetic data to enhance code repair capabilities for models (Allamanis, Jackson-Flux, and Brockschmidt 2021; Yasunaga and Liang 2021; He, Beurer-Kellner, and Vechev 2022). We leave this for future work to make the instructions generated by LLMs closer to the true distribution of instructions written by humans by combining VERSE and other methods.

5.4 Distillation to Smaller Models

We also investigate the possibility of condensing knowledge into smaller Code LLMs by training them with data generated by larger Code LLMs, which is known as knowledge distillation (Agarwal et al. 2023). We use Deepseek-Coder-1.3B-Base for experiments, comparing the results of self-improvement and distillation from Deepseek-Coder-6.7B-Base. Data is generated through the pipeline of VERSE. Results are presented in Figure 6. It can be found that the model trained using distilled data is better than the self-trained model. The evaluation metrics of the distillation model increase by 7.1% on average compared with the base model and increase by 2.5% compared with the self-trained model. This enlightens us that the distilled data, verified by the large model, can enhance the small model's ability to obtain more competitive models in scenarios with limited computing resources.

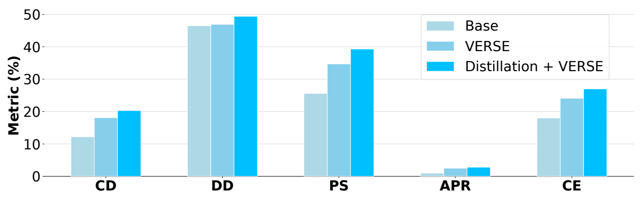


Figure 6: Distillation from large models to small models.

5.5 Iterative Optimization

Since VERSE can enhance responses to code instructions, we also explore whether it can iteratively improve the model. We conduct experiments using CodeLlama-7B, which iteratively performs four rounds of training, each round collecting 10,000 instances for training through the VERSE pipeline. The results are shown in Figure 7. Iterative training has obvious improvements in the first two epochs, but there is no obvious change in the last two epochs. This may be because the functional correctness of the responses is an absolute metric. Therefore, the degree of optimization has a certain upper limit, and it is difficult to carry out many rounds of optimization like relative metrics (Sun et al. 2023; Chen et al. 2024; Yuan et al. 2024).

5.6 Rerank with Verification

Similar to various reranking methods (Inala et al. 2022; Ni et al. 2023; Zhang et al. 2022), we also use the Python subset in HUMANEVALPACK to evaluate the model's ability to verify and rank multiple responses by assessing the execution results of the reranked responses. The verification and data ranking methods are the same as VERSE. Figure 8 presents the results. It can be found that reranking through self-verification has a good improvement in model performance, and the model trained with VERSE shows even greater improvement. The verification difficulty of different tasks varies and has a certain impact on the extent of improvement. For example, the instructions of fixing programs include error samples, and it is less difficult to verify whether the repair is successful, so the improvement through verification is

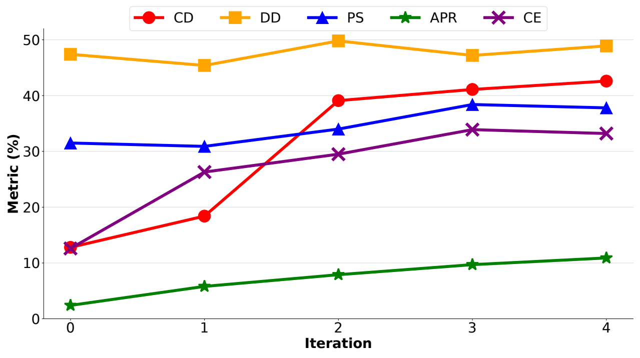


Figure 7: Changes in metrics of each task during the iterative training of CodeLlama-7B.

huge. For code explanation tasks, the improvement is smaller because of the uncertainty of regenerating the program for verification based on the generated explanation.

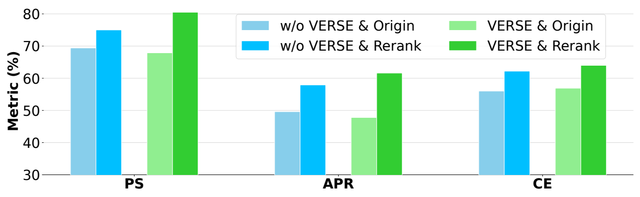


Figure 8: Evaluation results based on verfication and reranking for responses.

6 Limitation

In the experimental section, due to resource and data limitations, we do not conduct sufficient experiments on larger models. For our approach, a limitation is that VERSE depends on the execution of programs, and it is impossible to cover all possible situations; code instructions with complex dependencies or multiple file relationships are difficult to execute directly. In addition, VERSE enhances the functional correctness of responses through execution-based self-verification, but the evaluation metrics for code instructions are not unique. Besides ensuring functional correctness, we also want the results to be reliable, explainable, and faster. We will overcome these limitations and combine them with other methods to improve the quality of instructions in the future.

7 Conclusion

We proposed VERSE, utilizing verification-based self-play to enhance responses to code instructions. Our approach combined execution and self-consistency, allowing Code LLMs to self-verify the functional correctness of responses and train themselves based on the verification results. Our experiments succeeded on multiple Code LLMs, including CodeLlama, Deepseek-Coder, and various benchmarks such as CODEXGLUE and HUMANEVALPACK. They demonstrated that VERSE improves the model's ability and is easily generalized to different models and code-related tasks.

Acknowledgments

This research was supported by grants from the National Natural Science Foundation of China (Grants No. 62337001, 623B1020) and the Fundamental Research Funds for the Central Universities.

References

Agarwal, R.; Vieillard, N.; Zhou, Y.; Stanczyk, P.; Ramos, S.; Geist, M.; and Bachem, O. 2023. Generalized Knowledge Distillation for Auto-regressive Language Models. *arXiv preprint arXiv: 2306.13649*.

Allamanis, M.; Jackson-Flux, H.; and Brockschmidt, M. 2021. Self-Supervised Bug Detection and Repair. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*, volume 34, 27865–27876. Curran Associates, Inc.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; Guha, A.; Greenberg, M.; and Jangda, A. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Trans. Software Eng.*, 49(7): 3675–3691.

Chen, Z.; Deng, Y.; Yuan, H.; Ji, K.; and Gu, Q. 2024. Self-Play Fine-Tuning Converts Weak Language Models to Strong Language Models. *arXiv preprint arXiv: 2401.01335*.

Chung, H. W.; Hou, L.; Longpre, S.; Zoph, B.; Tay, Y.; Fedus, W.; Li, Y.; Wang, X.; Dehghani, M.; Brahma, S.; Webson, A.; Gu, S. S.; Dai, Z.; Suzgun, M.; Chen, X.; Chowdhery, A.; Castro-Ros, A.; Pellat, M.; Robinson, K.; Valter, D.; Narang, S.; Mishra, G.; Yu, A.; Zhao, V.; Huang, Y.; Dai, A.; Yu, H.; Petrov, S.; Chi, E. H.; Dean, J.; Devlin, J.; Roberts, A.; Zhou, D.; Le, Q. V.; and Wei, J. 2022. Scaling Instruction-Finetuned Language Models. *arXiv preprint arXiv: 2210.11416*.

Dao, T. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv preprint arXiv: 2307.08691*.

Dong, H.; Xiong, W.; Goyal, D.; Zhang, Y.; Chow, W.; Pan, R.; Diao, S.; Zhang, J.; SHUM, K.; and Zhang, T. 2023. RAFT: Reward rAnked FineTuning for Generative Foundation Model Alignment. *Transactions on Machine Learning Research*.

Gulwani, S.; Radicek, I.; and Zuleger, F. 2016. Automated Clustering and Program Repair for Introductory Programming Assignments. *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y. K.; Luo, F.; Xiong, Y.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *arXiv preprint arXiv: 2401.14196*.

Haluptzok, P.; Bowers, M.; and Kalai, A. T. 2023. Language Models Can Teach Themselves to Program Better. In *The Eleventh International Conference on Learning Representations*.

He, J.; Beurer-Kellner, L.; and Vechev, M. T. 2022. On Distribution Shift in Learning-based Bug Detectors. *International Conference on Machine Learning*.

Hong, S.; Zhuge, M.; Chen, J.; Zheng, X.; Cheng, Y.; Zhang, C.; Wang, J.; Wang, Z.; Yau, S. K. S.; Lin, Z.; Zhou, L.; Ran, C.; Xiao, L.; Wu, C.; and Schmidhuber, J. 2023. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. *arXiv preprint arXiv: 2308.00352*.

Hu, Y.; Ahmed, U. Z.; Mechtaev, S.; Leong, B.; and Roychoudhury, A. 2019. Re-factoring based Program Repair applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 388–398. IEEE/ACM.

Huang, B.; Lu, S.; Chen, W.; Wan, X.; and Duan, N. 2023. Enhancing Large Language Models in Coding Through Multi-Perspective Self-Consistency. *arXiv preprint arXiv: 2309.17272*.

Inala, J.; Wang, C.; Yang, M.; Codas, A.; Encarnaci'on, M.; Lahiri, S. K.; Musuvathi, M.; and Gao, J. 2022. Fault-Aware Neural Code Rankers. *Neural Information Processing Systems*.

Johnson, D. D.; Tarlow, D.; and Walder, C. 2023. R-U-SURE? Uncertainty-Aware Code Suggestions By Maximizing Utility Across Random User Intents. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, 15262–15306. PMLR.

Jung, J.; Qin, L.; Welleck, S.; Brahman, F.; Bhagavatula, C.; Bras, R. L.; and Choi, Y. 2022. Maieutic Prompting: Logically Consistent Reasoning with Recursive Explanations. *Conference on Empirical Methods in Natural Language Processing*.

Khakhar, A.; Mell, S.; and Bastani, O. 2023. PAC Prediction Sets for Large Language Models of Code. *International Conference on Machine Learning*.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J. E.; Zhang, H.; and Stoica, I. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. *Symposium on Operating Systems Principles*.

Li, R.; He, L.; Liu, Q.; Zhao, Y.; Zhang, Z.; Huang, Z.; Su, Y.; and Wang, S. 2024a. CONSIDER: Commonalities and Specialties Driven Multilingual Code Retrieval Framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 8679–8687.

Li, R.; Liu, Q.; He, L.; Zhang, Z.; Zhang, H.; Ye, S.; Lu, J.; and Huang, Z. 2024b. Optimizing Code Retrieval: High-Quality and Scalable Dataset Annotation through Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2053–2065.

Li, R.; Yin, Y.; Dai, L.; Shen, S.; Lin, X.; Su, Y.; and Chen, E. 2022. PST: measuring skill proficiency in programming exercise process via programming skill tracing. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2601–2606.

Liu, Z.; Zhuang, Y.; Liu, Q.; Li, J.; Zhang, Y.; Huang, Z.; Wu, J.; and Wang, S. 2024. Computerized Adaptive Testing via Collaborative Ranking. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Longpre, S.; Hou, L.; Vu, T.; Webson, A.; Chung, H. W.; Tay, Y.; Zhou, D.; Le, Q. V.; Zoph, B.; Wei, J.; and Roberts, A. 2023. The Flan Collection: Designing Data and Methods for Effective Instruction Tuning. *International Conference on Machine Learning*.

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C. B.; Drain, D.; Jiang, D.; Tang, D.; Li, G.; Zhou, L.; Shou, L.; Zhou, L.; Tufano, M.; Gong, M.; Zhou, M.; Duan, N.; Sundaresan, N.; Deng, S. K.; Fu, S.; and Liu, S. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *NeurIPS Datasets and Benchmarks*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv: 2306.08568*.

Mahbub, P.; Shuvo, O.; and Rahman, M. M. 2022. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. *International Conference on Software Engineering*.

Muennighoff, N.; Liu, Q.; Zebaze, A.; Zheng, Q.; Hui, B.; Zhuo, T. Y.; Singh, S.; Tang, X.; von Werra, L.; and Longpre, S. 2023. OctoPack: Instruction Tuning Code Large Language Models. *arXiv preprint arXiv: 2308.07124*.

Ni, A.; Iyer, S.; Radev, D.; Stoyanov, V.; Yih, W.-T.; Wang, S.; and Lin, X. V. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In Krause, A.; Brunskill, E.; Cho, K.; Engelhardt, B.; Sabato, S.; and Scarlett, J., eds., *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, 26106–26128. PMLR.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P. F.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 27730–27744. Curran Associates, Inc.

Rajbhandari, S.; Rasley, J.; Ruwase, O.; and He, Y. 2019. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. *International Conference for High Performance Computing, Networking, Storage and Analysis*.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; Kozhevnikov, A.; Evtimov, I.; Bitton, J.; Bhatt, M.; Ferrer, C. C.; Grattafiori, A.; Xiong, W.; Défossez, A.; Copet, J.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv: 2308.12950*.

Shazeer, N. M.; and Stern, M. 2018. Adafactor: Adaptive Learning Rates with Sublinear Memory Cost. *International Conference on Machine Learning*.

Sun, Z.; Shen, Y.; Zhou, Q.; Zhang, H.; Chen, Z.; Cox, D.; Yang, Y.; and Gan, C. 2023. Principle-Driven Self-Alignment of Language Models from Scratch with Minimal Human Supervision. *NEURIPS*.

Svajlenko, J.; Islam, J. F.; Keivanloo, I.; Roy, C. K.; and Mia, M. M. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, 476–480. IEEE.

Tang, Y.; Zhou, L.; Blanco, A.; Liu, S.; Wei, F.; Zhou, M.; and Yang, M. 2021. Grammar-Based Patches Generation for Automated Program Repair. In Zong, C.; Xia, F.; Li, W.; and Navigli, R., eds., *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 1300–1305. Online: Association for Computational Linguistics.

Taori, R.; Gulrajani, I.; Zhang, T.; Dubois, Y.; Li, X.; Guestrin, C.; Liang, P.; and Hashimoto, T. B. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; and Zhou, D. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *International Conference on Learning Representations*.

Wang, Y.; Kordi, Y.; Mishra, S.; Liu, A.; Smith, N. A.; Khashabi, D.; and Hajishirzi, H. 2023a. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In Rogers, A.; Boyd-Graber, J. L.; and Okazaki, N., eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, 13484–13508. Association for Computational Linguistics.

Wang, Y.; Le, H.; Gotmare, A. D.; Bui, N. D. Q.; Li, J.; and Hoi, S. C. H. 2023b. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *Conference on Empirical Methods in Natural Language Processing*.

Yasunaga, M.; and Liang, P. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. *International Conference on Machine Learning*.

Young, H.; Bastani, O.; and Naik, M. 2019. Learning Neurosymbolic Generative Models via Program Synthesis. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 7144–7153. PMLR.

Yuan, W.; Pang, R. Y.; Cho, K.; Sukhbaatar, S.; Xu, J.; and Weston, J. 2024. Self-Rewarding Language Models. *arXiv preprint arXiv: 2401.10020*.

Zhang, T.; Yu, T.; Hashimoto, T.; Lewis, M.; tau Yih, W.; Fried, D.; and Wang, S. I. 2022. Coder Reviewer Reranking for Code Generation. *International Conference on Machine Learning*.

Zhao, Y.; Huang, Z.; Ma, Y.; Li, R.; Zhang, K.; Jiang, H.; Liu, Q.; Zhu, L.; and Su, Y. 2024. RePair: Automated Program Repair with Process-based Feedback. *Annual Meeting of the Association for Computational Linguistics*.

Zhou, Y.; Liu, S.; Siow, J.; Du, X.; and Liu, Y. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, 10197–10207.