VisCoder: Fine-Tuning LLMs for Executable Python Visualization Code Generation

Anonymous ACL submission

Abstract

Large language models (LLMs) often struggle with visualization tasks like plotting diagrams, charts, where success depends on both code correctness and visual semantics. Existing instruction-tuning datasets lack execution-grounded supervision and offer limited support for iterative code correction, resulting in fragile and unreliable plot generation. We present **VisCode-200K**, a large-scale instruction tuning dataset for Python-based visualization and self-correction. It contains over 200K examples from two sources: (1) validated plotting code from open-source repositories, paired with natural language instructions and rendered plots; and (2) 45K multi-turn correction dialogues from Code-Feedback, enabling models to revise faulty code using runtime feedback. We fine-tune Qwen2.5-Coder-Instruct on VisCode-200K to create **VisCoder**, and evaluate it on PandasPlotBench. VisCoder significantly outperforms strong open-source baselines and approaches the performance of proprietary models like GPT-4o-mini. We further adopt a *self-debug* evaluation protocol to assess iterative repair, demonstrating the benefits of feedback-driven learning for executable, visually accurate code generation.

1 Introduction

Despite the growing capabilities of large language models (LLMs) in general-purpose code generation (Chen et al., 2021; Guo et al., 2024), they continue to struggle with one of the most common and visually essential tasks in data analysis: *generating code that produces a valid and semantically meaningful plot*. For example, given a tabular description, models may generate code that appears syntactically correct and invokes the appropriate libraries (Dibia, 2023; Xie et al., 2024). But when executed, the result is often broken: exceptions are raised, plots render blank or malformed, or the visual fails to reflect the intended semantics of the instruction (Chen et al., 2024; Yang et al., 2024; Galimzyanov et al., 2024).

These failures are not incidental: they reflect structural challenges in visualization code generation. Unlike standard text-to-code tasks, visualization requires grounding across three modalities: *natural language* (the user instruction), *data structure* (the tabular or other data input), and *visual output* (the rendered chart). Execution correctness is not binary; a script may run and still fail to convey the intended meaning. Visualization libraries such as `matplotlib` (Hunter, 2007), `seaborn` (Waskom, 2021), and `plotly` (Inc, 2015) further complicate the task, with API idiosyncrasies and intricate bindings between data, layout, and style.

Current instruction-tuning datasets do not meet the demands of this setting. Most lack explicit visual grounding, do not enforce runtime validation, and provide little to no supervision for recovery from failure. As a result, even advanced open models like Qwen-Coder (Hui et al., 2024) struggle with executable, semantically accurate visualization code, particularly when debugging is required (Zheng et al., 2024).

To address these gaps, we introduce **VisCode-200K**, a new instruction-tuning dataset for Python-based visualization code generation and multi-turn correction. VisCode-200K contains over 200K supervised examples derived from two complementary data sources: 1) **Executable visualization code**, extracted from open-source Python repositories and filtered across widely-used plotting libraries, including `matplotlib`, `seaborn` and others. All code samples are validated for runtime executability and paired with rendered plots. Natural language instructions are generated using LLMs conditioned on both the code and its output image (Galimzyanov et al., 2024). 2) **Multi-turn revision dialogues**, drawn from the Code-Feedback dataset (Zheng et al., 2024), which contains realistic interactions where models revise faulty

Python code based on runtime errors and follow-up prompts. While not visualization-specific, these traces provide essential supervision for teaching models to debug and recover from execution failures. This dual-source construction enables training for both *single-shot generation* and *multi-round refinement*, allowing models to generate code initially and improve it iteratively through feedback.

To demonstrate the effectiveness of VisCode-200K, we fine-tune Qwen2.5-Coder-Instruct (Hui et al., 2024) at both 3B and 7B scales to produce **VisCoder**, an open-source model tuned specifically for Python visualization tasks. We evaluate VisCoder on **PandasPlotBench** (Galimzyanov et al., 2024), a benchmark that assesses executable code generation from natural language and data previews across three plotting libraries. We also introduce a **self-debug evaluation mode**, in which models are given multiple rounds to revise failed outputs based on execution traces, simulating a realistic developer-style correction loop.

Our experiments show that VisCoder substantially outperforms competitive open-source baselines. VisCoder-3B and 7B achieve average execution pass rate improvements of **19.6** and **14.5** points over Qwen2.5-Coder. Under self-debug mode, it reaches over **90%** execution pass rate on `Matplotlib` and `Seaborn`. Compared to proprietary models, VisCoder-7B surpasses GPT-4o-mini on both `Seaborn` and `Plotly` under the default setting, and approaches GPT-4o performance on both libraries after self-debugging. At 3B scale, it outperforms GPT-4o-mini on `Seaborn` and narrows the gap in other libraries. These results demonstrate the impact of combining domain-specific instruction tuning with feedback-driven correction for grounded visualization code generation.

2 Related Work

LLMs for Visualization Code Generation. Recent work has explored using large language models to generate visualization code from natural language prompts. Benchmarks such as MatPlotAgent and VisEval (Yang et al., 2024; Chen et al., 2024) evaluate model performance on structured NL2VIS tasks with paired chart specifications and data previews, while PandasPlotBench (Galimzyanov et al., 2024) provides a curated benchmark for assessing executable visualization code generation across multiple plotting libraries. Plot2Code (Wu et al., 2024) investigates the reverse direction by generating code from rendered plots, but it relies on image-level inputs and bypasses the textual reasoning central to real-world data workflows. These studies highlight persistent challenges in semantic grounding, API correctness, and robustness across different plotting tasks. Broader evaluations have analyzed model behavior across visualization types and libraries (Vázquez, 2024; Podo et al., 2024), while specification-based approaches using Vega-Lite (Xie et al., 2024) offer an alternate formulation that lacks direct executability. Beyond evaluation, systems like LIDA (Dibia, 2023) and VisPath (Seo et al., 2025) incorporate summarization, code synthesis, and feedback-driven refinement into end-to-end pipelines. Related efforts have also extended visual code generation to structured domains such as parametric CAD modeling (Li et al., 2025) and mathematical animation (Ku et al., 2025), where outputs reflect domain-specific constraints rather than general-purpose charting semantics. However, most prior work lacks training data grounded in execution outcomes and provides limited support for iterative refinement. These limitations hinder model reliability, especially when generating code that must be both syntactically correct and semantically faithful to the intended visualization.

Execution Feedback and Code Correction. Execution feedback has been widely explored as a supervisory signal for improving the reliability of code generation. Prior work investigates using runtime traces to guide post-hoc refinement (Jain et al., 2025; Chen et al., 2025; Tian et al., 2024; Zhang and Yang, 2025), or integrates such signals into training through reinforcement learning (Gehring et al., 2024; Zeng et al., 2025). Other approaches emphasize multi-turn correction, where models revise faulty code using internal or external feedback (Madaan et al., 2023; Jiang et al., 2024; Zheng et al., 2024; Ruiz et al., 2025), or simulate debugging workflows with planning and agent collaboration (Grishina et al., 2025; Li et al., 2024). In the context of visualization, VisPath (Seo et al., 2025) and MatPlotAgent (Yang et al., 2024) explore chart refinement using visual feedback from rendered outputs. Yet despite these advances, supervision grounded in execution feedback or revision traces has rarely been used to train models for visualization code generation, where runtime validity and semantic alignment remain central challenges.

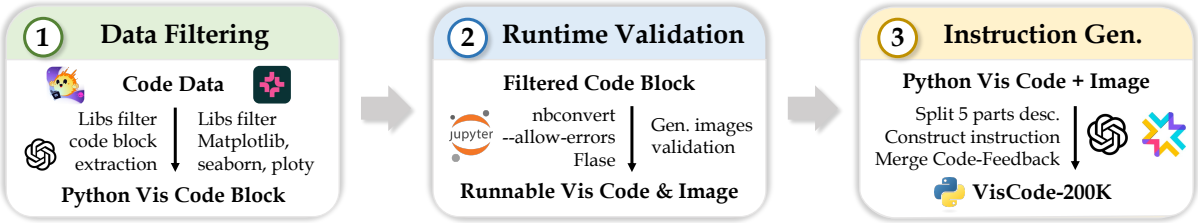


Figure 1: Data construction pipeline for VisCode-200K. We extract and filter visualization code blocks from open-source Python sources, validate their executability and plot rendering via Jupyter-based runtime checks, and generate structured instructions paired with rendered plots. We integrate multi-turn correction data from Code-Feedback during instruction construction to support iterative refinement.

3 VisCode-200K: A Python Visualization Instruction Tuning Dataset

In this section, we present **VisCode-200K**, a supervised instruction tuning dataset for Python-based visualization and feedback-driven code correction. It is designed to support robust code generation across diverse plotting libraries and to enable iterative refinement through multi-turn supervision.

VisCode-200K integrates two complementary sources of supervision. The first consists of executable visualization code extracted from open-source Python repositories, covering a wide range of real-world chart types, layouts, and plotting libraries. All samples are filtered to ensure runtime validity and compatibility with standard Python environments, exposing models to diverse and realistic plotting practices. The second source comprises multi-turn Python dialogues from the Code-Feedback dataset (Zheng et al., 2024), which offer supervision for revising faulty code in response to execution errors. While not specific to visualization, these interactions are critical for modeling realistic correction behaviors in iterative workflows.

Figure 1 provides an overview of the VisCode-200K construction pipeline, which consists of code filtering, runtime validation, and structured instruction generation. The following subsections detail each component.

3.1 Code Extraction from Public Repositories

To build a large corpus of executable Python visualization code, we source data from two open datasets: the Python subset of `stack-edu`[1] (Allal et al., 2025) and the chart/table partitions of `CoSyn-400K`[2] (Yang et al., 2025; Deitke et al., 2024). From these corpora, we extract code that uses commonly adopted visualization libraries, including `matplotlib`, `seaborn` and others, to ensure broad coverage of real-world plotting styles. The construction pipeline consists of four stages: library-based filtering, code block extraction, runtime validation, and instruction generation.

Filtering and Code Block Extraction. For the `stack-edu` source, which contains a large collection of Python code examples from educational contexts, we begin by applying library-based filters to identify approximately 1.7M samples that invoke common Python visualization libs. Since most examples embed visualization logic within broader program contexts, we use GPT-4o-mini (OpenAI, 2024a) to extract minimal, standalone plotting blocks. During this process, we inject mock data to replace missing inputs and ensure that each block can be executed in isolation. This structural cleaning step yields code samples that reflect realistic plotting usage while remaining compatible with our runtime pipeline. After filtering and reconstruction, we obtain roughly 1M candidate blocks. To balance library distribution, we retain all `seaborn` and ohter samples and randomly subsample a matching number of `matplotlib` examples, resulting in a curated subset of 300K visualization blocks.

From `CoSyn-400K`, we extract 112K Python code snippets that include calls to one of the target visualization libraries. CoSyn provides high-quality synthetic plotting code spanning a wide range of styles, with well-rendered outputs and consistent structure. Unlike `stack-edu`, it stores code and data separately, which requires reconstruction to enable runtime execution. We synthesize runnable scripts by inserting inline annotations such as column headers and the first data row to emulate realistic `pandas.read_csv` loading. When necessary, we append missing plotting function calls to ensure that each script can execute fully within a notebook environment.

[1] hf.co/datasets/HuggingFaceTB/stack-edu
[2] hf.co/datasets/allenai/CoSyn-400K

Runtime Validation. To verify executability, we run each code block in an isolated Jupyter environment using `nbconvert` with `allow-error=False`. We enforce a timeout and terminate executions that hang or enter infinite loops using a simulated keyboard interrupt. Only samples that run successfully and generate a valid image file are retained. This step yields 105K validated plotting scripts from `stack-edu` and 50K from `CoSyn-400K`, each paired with its corresponding output image.

Instruction Generation. To construct meaningful instructions for visualization code generation, we use GPT-4o (OpenAI, 2024b) to synthesize instruction components based on each validated code block and its corresponding plot. This enables the model to incorporate both structural code features and visual semantics from the rendered image.

Each instruction consists of five components: (1) a brief setup description specifying the programming language and visualization libraries used; (2) a data description summarizing the tabular input and column semantics; (3) a data block indicating the input table, either as mock data (for `stack-edu`) or a two-row preview (for `CoSyn`); (4) a high-level plot description outlining axes and structural layout; and (5) a style description capturing colors, grid layout, and other visual properties.

For `stack-edu` samples, mock data is extracted directly from the code block, where it was inserted during preprocessing. For `CoSyn`, where data is stored separately, we construct a compact preview using the first two rows of the table. The five components are then assembled using a fixed template to form the final instruction:

```
[Plot Description]
[Setup]
[Data Description]
"The mock data shows below:" or "The
first two rows of the data are shown
below:"
[Data]
[Plot Style Description]
```

This format enforces a consistent prompt structure across both data sources, providing models with a clear description of the target plot as well as the data and style required to render it.

3.2 Multi-turn Instruction-following Dialogues with Execution Feedback

To train models with self-correction capabilities, we incorporate 45K multi-turn dialogues from the `Code-Feedback`[3] dataset (Zheng et al., 2024). These dialogues involve Python-based tasks, including user instructions, model-generated code, and follow-up turns containing execution feedback or revision prompts.

We begin with 56K Python dialogues and remove those with excessive length or turn count to maintain consistency and reduce training complexity. The resulting 45K samples span diverse Python tasks with realistic correction behaviors.

While not specific to visualization, these dialogues offer valuable supervision for teaching models to revise faulty code based on runtime signals and to reason over iterative interactions. We integrate them into the instruction tuning corpus alongside the single-turn samples from `stack-edu` and `CoSyn`, enabling models to learn both initial generation and multi-turn refinement strategies.

[3] hf.co/datasets/m-a-p/Code-Feedback

4 Experiment Setup

Training Setup. We fine-tune Qwen2.5-Coder-Instruct (Hui et al., 2024) at two parameter scales: 3B and 7B. This allows us to assess the generalizability of VisCode-200K across different model capacities. Both models are trained for 3 epochs with a learning rate of 5×10^{-6}, a warm-up ratio of 0.05, and a cosine learning rate scheduler. We perform full-parameter tuning in bfloat16 precision on 8×A100 GPUs with a total batch size of 128.

Evaluation Setup. We evaluate models using PandasPlotBench (Galimzyanov et al., 2024), a benchmark designed to assess the ability of language models to generate executable and semantically accurate visualization code from tabular data descriptions. It contains 175 tasks spanning three widely used Python plotting libraries: `matplotlib`, `seaborn`, and `plotly`.

Each task includes a natural language instruction and a preview of the input DataFrame. The model is expected to generate Python code that produces a valid plot when executed according to the instruction. The benchmark reports three metrics: (1) *Incorrect Code Rate*, the proportion of outputs that fail to produce any plot; and two GPT-4o-judged scores: (2) a task-based score measuring alignment with the instruction, and (3) a visual score assessing similarity to the reference plot.

Among these metrics, Incorrect Code Rate provides only a coarse signal of success. It indicates whether a plot is rendered, but does not capture execution errors if a figure is produced. As a result, blank or semantically meaningless outputs—such

as plots with only axes—may be misclassified as correct. To address this issue, we introduce an additional metric: **Execution Pass Rate**, defined as the percentage of outputs that execute without error.

Self-Debug Evaluation Mode. To evaluate a model's ability to recover from failure, we extend the benchmark with a *self-debug evaluation mode*. In this setting, if the initial generation fails to execute or does not produce a valid plot, the model is allowed up to K rounds to iteratively revise its output based on accumulated feedback.

At each round, only the tasks that remain unsolved from the previous attempt are reconsidered. The model receives a multi-turn prompt constructed as a dialogue, including the original instruction, its failed code response, and a follow-up message requesting correction based on the execution error. Conditioned on this dialogue history, the model generates a revised version of the code. Tasks are considered successfully fixed if the generated code executes without error and produces a valid plot. These tasks are excluded from subsequent rounds.

Algorithm 1 Self-Debug Evaluation Protocol

```
Let F_0 be failed tasks from initial evaluation
for i = 1 to K do
    for each task x in F_{i-1} not yet fixed do
        Fix x via feedback-driven prompting
        Evaluate the result of the revised code
        if successful then
            Mark x as fixed & record output
        else
            Record x's latest failed output
        end if
    end for
end for
Evaluate all tasks with final recorded outputs
```

We set $K = 3$ for all experiments. After the final round of self-debug, each task is evaluated based on its recorded final output, which is either the successfully revised version from an earlier round or the last failed attempt if no fix was found. The resulting outputs are scored using the same evaluation pipeline as in the default setting. The full procedure is summarized in Algorithm 1.

This iterative process simulates a developer-style debugging loop and enables systematic evaluation of the model's ability to recover from failure through multi-round code correction.

5 Main Results

We present the main experimental results on PandasPlotBench, including overall model comparisons, performance under the self-debug evaluation protocol, error type analysis, and a training data ablation study.

5.1 Overall Model Comparison

We evaluate VisCoder models against both proprietary and open-source language models to assess executable visualization performance across scales and libraries. The proprietary group includes GPT-4o (OpenAI, 2024b), the strongest model in the original PandasPlotBench benchmark, and its lightweight variant GPT-4o-mini (OpenAI, 2024a). Among open-source baselines, we compare LLaMA-3.2-3B, LLaMA-3.1-8B (Grattafiori et al., 2024), Qwen2.5-Instruct, and Qwen2.5-Coder-Instruct (Team, 2024; Hui et al., 2024), evaluated at both 3B and 7B scales. VisCoder models are trained on VisCode-200K and fine-tuned using the same instruction tuning setup.

Table 1 summarizes model performance across the three plotting libraries. The following analysis focuses on execution success, task alignment, and visual fidelity, highlighting VisCoder's comparative strengths and remaining challenges.

Proprietary Models Remain Stronger. Proprietary models outperform open-source models by a wide margin across all plotting libraries. GPT-4o achieves the highest execution pass rates and the strongest judge-based scores, followed by its lightweight variant GPT-4o-mini. These results indicate more reliable execution and better semantic alignment with task instructions, particularly in complex visualization settings. In contrast, open-source models such as LLaMA and Qwen2.5-Instruct consistently underperform across all metrics. This reinforces the gap between proprietary and open-source systems on execution-sensitive and semantically grounded code generation.

Plotly Presents Harder Challenge. Performance differs across plotting libraries. While most models perform reliably on `matplotlib` and `seaborn`, results on `plotly` are markedly lower, especially for open-source models. Execution pass rates often fall below 35%, and task and visual scores drop accordingly. Generated plots frequently fail to reflect the intended semantics or produce complete visuals. This suggests that `plotly`'s

Model	Matplotlib					Seaborn					Plotly				
	Exec	Mean		Good(≥75)		Exec	Mean		Good(≥75)		Exec	Mean		Good(≥75)	
	Pass	vis	task	vis	task	Pass	vis	task	vis	task	Pass	vis	task	vis	task
GPT-4o	94.9	75	90	67%	93%	83.4	65	78	59%	80%	77.7	55	68	50%	70%
GPT-4o + Self Debug	**99.4**	77	93	69%	96%	**92.6**	69	84	63%	86%	**97.7**	68	84	61%	83%
GPT-4o-mini	88.6	68	86	59%	86%	62.3	45	57	41%	57%	69.1	48	52	42%	51%
GPT-4o-mini + Self Debug	97.7	72	92	65%	94%	72.0	47	60	43%	61%	**97.7**	62	71	51%	67%
∼ 3B Scale															
Llama-3.2-3B-Ins.	65.1	43	60	34%	55%	30.9	18	24	14%	21%	13.1	8	8	7%	8%
Qwen-2.5-3B-Ins.	74.3	55	68	49%	66%	58.3	43	58	33%	51%	30.9	19	23	17%	21%
Qwen-2.5-Coder-3B-Ins.	71.4	56	72	50%	69%	58.3	44	55	36%	51%	27.4	17	19	17%	18%
VisCoder-3B	81.7	60	69	53%	69%	73.7	48	65	38%	61%	60.6	38	45	32%	44%
VisCoder-3B + Self Debug	**85.1**	60	70	53%	69%	**78.3**	48	66	37%	62%	**64.6**	40	48	34%	47%
∼ 7B Scale															
Llama-3.1-8B-Ins.	81.1	61	76	51%	74%	65.7	51	64	45%	63%	30.9	21	22	20%	21%
Qwen2.5-7B-Ins.	77.1	64	76	53%	75%	66.3	51	63	46%	62%	56.0	38	42	31%	40%
Qwen2.5-Coder-7B-Ins.	78.3	63	76	58%	75%	68.6	51	63	40%	62%	48.0	29	34	24%	31%
VisCoder-7B	87.4	66	78	60%	80%	76.6	57	70	50%	68%	74.3	48	60	41%	61%
VisCoder-7B + Self Debug	**91.4**	67	81	62%	83%	**90.3**	62	77	51%	75%	**81.7**	51	65	44%	65%

Table 1: Performance of selected models on the PandasPlotBench benchmark. For each model, we report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**). The best-performing model in each scale is shown in **bold**, and the second best is underlined.

verbose syntax and less represented API structure pose greater challenges for current models.

VisCoder Closes the Open-Source Gap. VisCoder models consistently outperform their untuned Qwen2.5-Coder baselines across all libraries. At 3B, VisCoder improves both execution success and semantic alignment, with larger gains on `plotly` and `seaborn`, where baseline generations often fail to capture visual intent. At 7B, VisCoder outperforms GPT-4o-mini on both `seaborn` and `plotly`, while remaining slightly behind on `matplotlib`. These results demonstrate that domain-specific instruction tuning improves functional reliability and output fidelity, especially in libraries with more complex plotting structures.

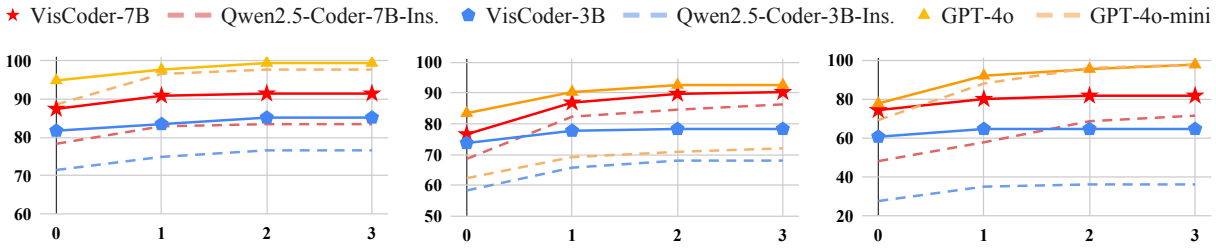
Self-Debug Further Boosts Performance. GPT-4o demonstrates strong self-debugging ability, reaching near-perfect execution pass rates after multiple rounds of correction. VisCoder models also improve substantially under this protocol. VisCoder-7B surpasses 90% execution success on both `matplotlib` and `seaborn`, with especially large gains on the latter. Task and visual scores improve consistently across rounds. These results show that VisCoder can generalize from its training data to refine failed outputs over multiple attempts, even without task-specific debugging supervision.

5.2 Self-Debug Evaluation Results

To analyze the dynamics of self-debugging, we track execution pass rates over multiple correction rounds by evaluating GPT-4o and GPT-4o-mini as proprietary baselines, alongside VisCoder models at 3B and 7B scales. To isolate the effects of instruction tuning, we also include untuned Qwen2.5-Coder models at matching sizes. Figure 2 shows execution pass rates from the initial generation (Attempt 0) through three rounds of self-debugging (Attempts 1–3), presented separately for each plotting library. Detailed breakdown of pass rates per model and library is provided in Appendix B.

Self-debug is broadly effective. Execution pass rates increase steadily over self-debug rounds for most models and libraries, indicating the overall effectiveness of the protocol. The first attempt typically yields the largest improvement, with smaller gains in subsequent rounds. This pattern suggests that a simple retry mechanism informed by execution feedback can recover a substantial portion of initial failures.

VisCoder yields stable behavior. Compared to their Qwen2.5-Coder baselines, VisCoder models show smaller per-round gains in execution pass rate but consistently achieve higher final performance. This suggests that VisCoder tends to generate stronger initial outputs and applies more stable corrections across rounds. The effect is most pro-



(a) Matplotlib (b) Seaborn (c) Plotly

Figure 2: **Execution pass rate** across self-debug rounds (Attempt 0–3), shown separately for three plotting libraries. Attempt 0 corresponds to the default output, while Attempts 1–3 represent subsequent correction rounds. Model groups are color-coded, with solid and dashed lines used to distinguish paired models. VisCoder models improve consistently across rounds, with VisCoder-7B gradually closing the gap to GPT-4o on `seaborn`. Y-axis ranges are scaled per subplot to match library-specific score distributions.

nounced with VisCoder-7B on `seaborn`, where execution rates increase steadily and approach GPT-4o by the final attempt.

Failures remain across models. Even the strongest model GPT-4o does not reach perfect execution rates after self-debugging. On `seaborn`, its performance plateaus after three rounds, leaving a non-trivial portion of failures unresolved. In contrast, VisCoder-3B stands out among small-scale models. It surpasses GPT-4o-mini on `seaborn` and performs competitively across other libraries. Meanwhile, we observe that smaller models tend to reach their performance ceiling more quickly, exhibiting smoother but more limited improvements across rounds.

5.3 Error Analysis

To examine the error recovery behavior of VisCoder-7B, we analyze how execution error counts transition before and after self-debugging. Table 2 summarizes four representative error types, grouped by plotting library. A detailed breakdown by model and debug round is provided in Appendix C.

Error Type	Matplotlib	Seaborn	Plotly
AttributeError	$5 \rightarrow 2$	$15 \rightarrow 2$	$5 \rightarrow 1$
TypeError	$7 \rightarrow 5$	$8 \rightarrow 4$	$3 \rightarrow 1$
KeyError	$1 \rightarrow 1$	$0 \rightarrow 0$	$1 \rightarrow 1$
ValueError	$4 \rightarrow 5$	$8 \rightarrow 7$	$29 \rightarrow 23$

Table 2: Execution error count transitions for VisCoder-7B across four representative error types, segmented by plotting library. Each value shows the transition from the initial to the post-debugging error count (X $\rightarrow$ Y).

Effective Recovery from Structural Errors. VisCoder-7B demonstrates strong self-correction ability on shallow, structural errors. *AttributeErrors* in Seaborn are reduced from 15 to 2, and *TypeErrors* in Plotly from 3 to 1. These failures typically result from incorrect method calls, invalid argument types, or simple syntax mistakes, and are often accompanied by clear diagnostic messages. As illustrated in Figure 4 and Figure 6, VisCoder can reliably correct such cases using runtime feedback, frequently producing valid plots on retry.

Persistent Failures in Semantic Execution Errors. Semantic execution errors such as *KeyError* and *ValueError* remain difficult to resolve (Figure 8). On Plotly, *ValueErrors* decrease from 29 to 23 across three rounds of correction, but a substantial number still remain. Meanwhile, *KeyErrors* show no improvement, remaining at 1 throughout. These failures are often caused by invalid trace configurations or mismatched array lengths and typically require reasoning over the input data structure. However, the model does not dynamically reassess the DataFrame during self-debug, leading to retries that rely on faulty assumptions. Compared to structural errors, semantic failures are less localized and more difficult to resolve through symbolic correction alone.

5.4 Training Data Ablation

We assess the contribution of each training data source in VisCode-200K through a controlled ablation study, including two reference points: the model trained on the full VisCode-200K dataset and the untuned Qwen2.5-Coder-7B-Instruct baseline. Separate Qwen2.5-Coder-7B models are fine-tuned on subsets from `stack-edu`, `CoSyn-400K`, and `Code-Feedback`, using the same instruction

tuning setup as the full configuration. All models are evaluated on PandasPlotBench under both default and self-debug modes. Table 3 shows execution pass rates across the three plotting libraries.

Model	Self-Debug	Matplotlib	Seaborn	Plotly
Qwen2.5-Coder-7B-Ins	✗	78.3	68.6	48.0
	✔	83.4	86.3	71.4
+ Stack-Edu-105K	✗	66.3	55.4	49.7
	✔	72.0	69.7	61.1
+ CoSyn-50K*	✗	0.0	0.0	5.7
	✔	0.0	0.0	6.3
+ Code-Feedback-45K	✗	88.0	44.0	62.9
	✔	90.9	59.4	77.7
+ VisCode-200K	✗	87.4	76.6	74.3
	✔	91.4	90.3	81.7

Table 3: Execution pass rates of Qwen2.5-Coder-7B models trained on individual subsets of VisCode-200K. Each model is evaluated across three libraries under both default (✗) and self-debug (✔) modes.

Stack-Edu provides moderate generalization. Using the subset from `stack-edu` results in modest gains over the baseline in `plotly` under the default setting (+1.7), but leads to significant drops on `matplotlib` and `seaborn` (–12.0 and –13.2). Self-debug improves pass rates across all libraries compared to their respective defaults, yet all scores remain below the untuned baseline. These results suggest that while stack-edu offers broad task coverage, it lacks the structural supervision and feedback-guided correction patterns needed for robust generalization.

CoSyn fails to generalize. The subset from `CoSyn-400K` fails to support effective instruction tuning for this task. Execution pass rates remain near zero across all libraries, and self-debug yields no meaningful improvement. Generated outputs often exhibit decoding instability, including repeated sequences, empty completions, or irrelevant boilerplate. A key reason is the homogeneous structure of the source data: all samples follow a fixed format consisting of imports, function definitions, and single function calls, which severely limits structural diversity during training. Combined with the synthetic and non-executable nature of the examples, this makes the single CoSyn subset ill-suited for executable visualization code generation.

Code-Feedback enhances structure but lacks breadth. The subset from `Code-Feedback` improves execution reliability on `matplotlib` and `plotly` in the default setting, outperforming the baseline by 9.7 and 14.9 points, respectively. These gains suggest that examples grounded in execution feedback help the model generate structurally valid and complete code. However, performance on `seaborn` remains low (44.0), and gains on `plotly` are limited compared to the full model. This reflects the general-purpose nature of the source data, which is not designed for visualization and lacks the task-specific grounding needed for broader transfer. Self-debug improves pass rates across libraries, but overall performance remains below that achieved with our full VisCode-200K dataset.

Full data offers complementary gains. The full VisCode-200K dataset yields the most consistent execution improvements across all plotting libraries and evaluation modes. Its performance under self-debug is particularly robust, with high pass rates maintained across structurally diverse tasks. These results reinforce the importance of domain-specific instruction tuning and multi-turn correction data for building reliable visualization-capable models.

6 Conclusion

In conclusion, VisCode-200K provides a large-scale instruction tuning dataset for Python visualization code generation, combining executable plotting examples with multi-turn correction dialogues grounded in runtime feedback. To validate its effectiveness, we evaluate VisCoder models on PandasPlotBench using the default setting. Additionally, We propose a self-debug protocol to simulate realistic correction workflows and assess model performance in this extended evaluation mode.

Experiments show that VisCoder substantially outperforms strong open-source baselines across execution and alignment metrics, and narrows the gap to proprietary models like GPT-4o-mini. Gains are particularly pronounced in settings that involve complex visualization structures, such as Plotly, and iterative correction through self-debugging. Ablation studies further demonstrate that structurally diverse, executable training data and feedback-driven supervision contribute to more robust performance across plotting libraries.

Looking forward, this work reinforces the importance of domain-specific instruction tuning and multi-turn correction supervision for building robust and semantically grounded visualization-capable models. Future extensions may explore broader plotting libraries, richer correction supervision, and evaluation methods that measure models' abilities to recover from execution errors.

Limitations

Although VisCoder substantially improves visualization code generation, its scope is currently limited to Python, leaving visualization tasks involving other programming languages such as R and JavaScript unexplored. Even within Python, performance on Plotly remains comparatively weaker due to its verbose syntax and complex API structure, frequently causing semantic execution errors that the existing self-debugging routine struggles to address. Furthermore, our evaluation relies on the default automatic judge model adopted from prior studies, without an independent analysis of its potential biases or reliability.

References

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, and 3 others. 2025. Smollm2: When smol goes big – data-centric training of a small language model.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Nan Chen, Yuge Zhang, Jiahang Xu, Kan Ren, and Yuqing Yang. 2024. Viseval: A benchmark for data visualization in the era of large language models. *IEEE Transactions on Visualization and Computer Graphics*.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. 2025. Revisit self-debugging with self-generated tests for code generation. *ArXiv preprint*, abs/2501.12793.

Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, and 1 others. 2024. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *ArXiv preprint*, abs/2409.17146.

Victor Dibia. 2023. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *ArXiv preprint*, abs/2303.02927.

Timur Galimzyanov, Sergey Titov, Yaroslav Golubev, and Egor Bogomolov. 2024. Drawing pandas: A benchmark for llms in generating plotting code. *ArXiv preprint*, abs/2412.02764.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *ArXiv preprint*, abs/2410.02089.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *ArXiv preprint*, abs/2407.21783.

Anastasiia Grishina, Vadim Liventsev, Aki Härmä, and Leon Moonen. 2025. Fully autonomous programming using iterative multi-agent debugging with large language models. *ACM Transactions on Evolutionary Learning*, 5(1):1–37.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *ArXiv preprint*, abs/2401.14196.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *ArXiv preprint*, abs/2409.12186.

John D Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95.

Plotly Technologies Inc. 2015. Collaborative data science. *Montreal: Plotly Technologies Inc Montral*, 376.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. 2025. Multi-turn code generation through single-step rewards. *ArXiv preprint*, abs/2502.20380.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Ledex: Training llms to better self-debug and explain code. *Advances in Neural Information Processing Systems*, 37:35517–35543.

Max Ku, Thomas Chong, Jonathan Leung, Krish Shah, Alvin Yu, and Wenhu Chen. 2025. Theoremexplainagent: Towards multimodal explanations for llm theorem understanding. *ArXiv preprint*, abs/2502.19400.

Jiahao Li, Weijian Ma, Xueyang Li, Yunzhong Lou, Guichun Zhou, and Xiangdong Zhou. 2025. Cad-llama: Leveraging large language models for computer-aided design parametric 3d model generation. *ArXiv preprint*, abs/2505.04481.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Codetree: Agent-guided tree search for code generation with large language models. *ArXiv preprint*, abs/2411.04329.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

OpenAI. 2024a. Gpt-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

OpenAI. 2024b. Hello gpt4-o. https://openai.com/index/hello-gpt-4o/.

Luca Podo, Muhammad Ishmal, and Marco Angelini. 2024. Vi (e) va llm! a conceptual stack for evaluating and interpreting generative ai-based visualizations. *ArXiv preprint*, abs/2402.02167.

Fernando Vallecillos Ruiz, Max Hort, and Leon Moonen. 2025. The art of repair: Optimizing iterative program repair with instruction-tuned models. *ArXiv preprint*, abs/2505.02931.

Wonduk Seo, Seungyong Lee, Daye Kang, Zonghao Yuan, and Seunghyun Lee. 2025. Vispath: Automated visualization code synthesis via multi-path reasoning and feedback-driven optimization. *ArXiv preprint*, abs/2502.11140.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and 1 others. 2024. Debugbench: Evaluating debugging capability of large language models. *ArXiv preprint*, abs/2401.04621.

Pere-Pau Vázquez. 2024. Are llms ready for visualization?

Michael L Waskom. 2021. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. 2024. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *ArXiv preprint*, abs/2405.07990.

Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14.

Yue Yang, Ajay Patel, Matt Deitke, Tanmay Gupta, Luca Weihs, Andrew Head, Mark Yatskar, Chris Callison-Burch, Ranjay Krishna, Aniruddha Kembhavi, and 1 others. 2025. Scaling text-rich image understanding via code-guided synthetic multimodal data generation. *ArXiv preprint*, abs/2502.14846.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, and 1 others. 2024. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *ArXiv preprint*, abs/2402.11453.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv preprint*, abs/2502.01718.

Xuanyu Zhang and Qing Yang. 2025. Extracting the essence and discarding the dross: Enhancing code generation with contrastive execution feedback. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 10569–10575.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *ArXiv preprint*, abs/2402.14658.

Table of Contents in Appendix

A Prompts Used for Dataset Construction **12**

B Breakdown Results in Self-Debug Mode Evaluation **13**
B.1 Matplotlib . . . 13
B.2 Seaborn . . . 13
B.3 Plotly . . . 14

C Breakdown Results by Error Type **15**
C.1 VisCoder Series . . . 15
C.2 GPT Series . . . 16
C.3 Qwen2.5 Series . . . 17
C.4 LLaMA Series . . . 19

D Case Study **20**
D.1 Matplotlib: Successful Generation . . . 20
D.2 Matplotlib: Self-Debug Recovery . . . 21
D.3 Seaborn: Successful Generation . . . 22
D.4 Seaborn: Self-Debug Recovery . . . 23
D.5 Plotly: Successful Generation . . . 24
D.6 Plotly: Self-Debug Recovery . . . 25

E Ethics and Reproducibility Statements **26**
E.1 Potential Risks . . . 26
E.2 Discuss the License for Artifacts . . . 26
E.3 Artifact Use Consistent With Intended Use . . . 26
E.4 Data Contains Personally Identifying Info or Offensive Content . . . 26
E.5 Documentation of Artifacts . . . 26
E.6 Parameters for Packages . . . 26
E.7 AI Assistants in Research or Writing . . . 26

A Prompts Used for Dataset Construction

In this section, we present the system prompts used during the construction of VisCode-200K. These prompts guide the automatic extraction of standalone visualization code from mixed-context sources, and support the generation of structured natural language instructions aligned with rendered plots.

Code Extraction Prompt

Model: GPT-4o-mini

You are a Python code extraction agent.

Given a Python code snippet and the used library, your task is to extract a self-contained and **runnable Python code block** that demonstrates how the specified library is actually used in the original code.
Use mock data where needed (e.g., pandas DataFrame, NumPy arrays), but keep it minimal and logically aligned with the original usage. Retain any important structure, function calls, or plotting styles that reflect meaningful usage of the library.
- Do not include ‘plt.close()‘ or similar calls.
- If the library is only imported but never used, or if there is insufficient information to construct a meaningful runnable code block, return "null" (a literal string).
- Return only the Python code block enclosed in triple backticks like this: “‘python ... “‘, with nothing else.

Used Library: `{used_libs}`
Code: `{code}`

Instruction Generation Prompt: `stack-edu`

Model: GPT-4o

Write the general TASK to write a code for plotting the given mock data.

The code with mock data is given below, and the result of the generated plot image is given at the end.

Split task into five parts:
1. Setup (describe programming language and libraries required to generate the plot).
2. Data Description (some short description of the mock data).
3. Data Generation (the data-generation lines copied verbatim).
4. Plot Description (describe the structural layout of the plot, without referencing libraries or function names. Begin with “Generate...” or “Create...”).
5. Plot Style Description (describe the visual styling aspects of the plot, without referencing libraries or function).

CODE: `{code}`

Each part of the task must start on a new line, numbered 1 through 5. Use plain text only. Do not include any markdown symbols.

Instruction Generation Prompt: `CosyN-400K`

Model: GPT-4o

Write the general TASK to write a code for plotting the given data.

The top two rows of the data are included in the code comments, showing the CSV structure, and the result of the generated plot image is given at the end.

Split task into four parts:
1. Setup (describe programming language and libraries required to generate the plot).
2. Data Description (some short description of the given data).
3. Plot Description (describe the structural layout of the plot, without referencing libraries or function names. Begin with “Generate...” or “Create...”).
4. Plot Style Description (describe the visual styling aspects of the plot, without referencing libraries or function).

CODE: `{code}`

Each part of the task must start on a new line, numbered 1 through 4. Use plain text only. Do not include any markdown symbols.

B Breakdown Results in Self-Debug Mode Evaluation

In this section, we provide a breakdown of model performance under the self-debug setting. For each visualization library, we report execution pass rates across up to three rounds of automatic correction, grouped by model series.

B.1 Matplotlib

Model	Normal	Self-Debug Attempt		
		Round 1	Round 2	Round 3
GPT-4o	94.9	97.7	99.4	99.4
GPT-4o-mini	88.6	96.6	97.7	97.7
Llama-3.2-3B-Instruct	65.1	76.6	80.0	81.7
Qwen2.5-3B-Instruct	74.3	79.4	82.9	84.6
Qwen2.5-Coder-3B-Instruct	71.4	74.9	76.6	76.6
VisCoder-3B	81.7	83.4	85.1	85.1
Llama-3.1-8B-Instruct	81.1	89.7	92.6	93.7
Qwen2.5-7B-Instruct	77.1	83.4	88.0	89.7
Qwen2.5-Coder-7B-Instruct	78.3	82.9	83.4	83.4
VisCoder-7B	87.4	90.9	91.4	91.4

Table 4: Execution pass rates (%) on Matplotlib tasks under the normal and self-debug settings. Models that fail initially are allowed up to three rounds of automatic correction.

[Back to Appendix Contents]

B.2 Seaborn

Model	Normal	Self-Debug Attempt		
		Round 1	Round 2	Round 3
GPT-4o	83.4	90.3	92.6	92.6
GPT-4o-mini	62.3	69.1	70.9	72.0
Llama-3.2-3B-Instruct	30.9	64.6	72.0	74.9
Qwen2.5-3B-Instruct	58.3	64.0	73.7	75.4
Qwen2.5-Coder-3B-Instruct	58.3	65.7	68.0	68.0
VisCoder-3B	73.7	77.7	78.3	78.3
Llama-3.1-8B-Instruct	65.7	78.9	84.6	90.3
Qwen2.5-7B-Instruct	66.3	79.4	85.7	89.7
Qwen2.5-Coder-7B-Instruct	68.6	82.3	84.6	86.3
VisCoder-7B	76.6	86.9	89.7	90.3

Table 5: Execution pass rates (%) on Seaborn tasks under the normal and self-debug settings. All models undergo up to three rounds of automatic correction after initial failure.

[Back to Appendix Contents]

B.3 Plotly

Model	Normal	Self-Debug Attempt		
		Round 1	Round 2	Round 3
GPT-4o	77.7	92.0	95.4	97.7
GPT-4o-mini	69.1	88.0	96.0	97.7
Llama-3.2-3B-Instruct	13.1	20.6	24.0	28.0
Qwen2.5-3B-Instruct	30.9	36.0	42.3	48.0
Qwen2.5-Coder-3B-Instruct	27.4	34.9	36.0	36.0
VisCoder-3B	60.6	64.6	64.6	64.6
Llama-3.1-8B-Instruct	30.9	43.4	53.7	58.3
Qwen2.5-7B-Instruct	56.0	66.3	72.6	77.1
Qwen2.5-Coder-7B-Instruct	48.0	57.7	68.6	71.4
VisCoder-7B	74.3	80.0	81.7	81.7

Table 6: Execution pass rates (%) on Plotly tasks under the normal and self-debug settings. All models undergo up to three rounds of automatic correction after initial failure.

[Back to Appendix Contents]

C Breakdown Results by Error Type

In this section, we provide a detailed breakdown of execution error types across model families, plotting libraries, and self-debugging rounds. For each model series, we report the number of Python exceptions observed under default execution and across up to three rounds of automatic correction.

C.1 VisCoder Series

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	**Round 1**	**Round 2**	**Round 3**	**Normal**	**Round 1**	**Round 2**	**Round 3**	**Normal**	**Round 1**	**Round 2**	**Round 3**
AttributeError	5	2	2	2	15	3	2	2	5	1	1	1
AxisError	-	-	-	-	1	1	1	1	-	-	-	-
ImportError	1	0	0	0	1	0	0	0	-	-	-	-
IndexError	1	0	0	0	-	-	-	-	1	1	1	1
KeyError	1	2	1	1	-	-	-	-	0	1	1	2
KeyboardInterrupt	1	1	1	1	2	1	1	1	1	1	1	1
NameError	-	-	-	-	5	4	2	1	-	-	-	-
OSError	1	1	1	1	1	1	1	1	1	1	1	1
SyntaxError	1	0	0	0	-	-	-	-	5	3	2	2
TypeError	7	5	5	5	8	5	4	4	3	1	1	1
ValueError	4	5	5	5	8	8	7	7	29	26	24	23
Total Errors	22	16	15	15	41	23	18	17	45	35	32	32

Table 7: Distribution of execution errors for VisCoder-7B across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	**Round 1**	**Round 2**	**Round 3**	**Normal**	**Round 1**	**Round 2**	**Round 3**	**Normal**	**Round 1**	**Round 2**	**Round 3**
AttributeError	7	3	2	2	20	10	10	9	4	4	3	2
ImportError	-	-	-	-	-	-	-	-	1	1	1	0
IndexError	2	2	2	2	4	3	3	3	1	1	1	1
KeyError	2	3	3	3	3	4	4	4	-	-	-	-
KeyboardInterrupt	0	0	1	4	2	2	2	3	5	4	4	29
NameError	0	1	0	0	1	1	1	2	0	2	0	0
OSError	-	-	-	-	1	1	1	1	-	-	-	-
SyntaxError	3	3	2	0	1	1	1	0	8	6	6	1
TypeError	5	5	4	4	2	4	3	3	9	6	6	6
ValueError	13	12	12	11	12	13	13	13	41	38	41	23
Total Errors	32	29	26	26	46	39	38	38	69	62	62	62

Table 8: Distribution of execution errors for VisCoder-3B across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

C.2 GPT Series

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	-	-	-	-	-	-	-	-	4	1	0	0
Exception	-	-	-	-	1	0	0	0	1	0	0	0
IndexError	1	0	0	0	1	1	0	0	-	-	-	-
KeyError	-	-	-	-	-	-	-	-	1	1	0	0
KeyboardInterrupt	-	-	-	-	-	-	-	-	2	1	2	2
ModuleNotFoundError	1	0	0	0	1	0	0	0	-	-	-	-
NameError	-	-	-	-	14	12	13	13	2	0	0	0
RuntimeError	-	-	-	-	1	0	0	0	-	-	-	-
SyntaxError	-	-	-	-	-	-	-	-	2	0	0	0
TypeError	2	1	0	0	6	1	0	0	3	1	1	0
ValueError	5	3	1	1	5	3	0	0	24	10	5	2
Total Errors	9	4	1	1	29	17	13	13	39	14	8	4

Table 9: Distribution of execution errors for GPT-4o across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	3	0	0	0	2	0	0	0	13	2	0	0
Exception	1	0	0	0	2	0	1	0	1	1	1	1
FileNotFoundError	1	0	0	0	1	1	0	0	-	-	-	-
ImportError	1	0	0	0	-	-	-	-	-	-	-	-
IndexError	0	1	1	1	1	1	1	1	0	2	1	1
KeyError	2	1	0	0	1	1	0	0	-	-	-	-
KeyboardInterrupt	-	-	-	-	1	0	0	0	1	0	0	0
ModuleNotFoundError	1	0	0	0	-	-	-	-	2	0	0	0
NameError	1	0	0	0	43	48	47	47	11	0	0	0
TypeError	1	1	1	0	4	1	0	0	5	1	1	0
ValueError	9	3	2	3	11	2	2	1	21	15	4	2
Total Errors	20	6	4	4	66	54	51	49	54	21	7	4

Table 10: Distribution of execution errors for GPT-4o-mini across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

C.3 Qwen2.5 Series

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	12	9	9	9	17	8	7	7	8	5	3	3
FileNotFoundError	-	-	-	-	0	1	1	1	-	-	-	-
ImportError	1	0	0	0	-	-	-	-	-	-	-	-
IndexError	1	1	1	1	1	0	0	0	1	1	1	1
KeyError	3	4	3	3	2	1	1	1	14	17	0	0
KeyboardInterrupt	1	1	1	1	1	2	2	2	2	1	1	1
ModuleNotFoundError	-	-	-	-	-	-	-	-	0	1	1	1
NameError	1	0	0	0	17	8	5	3	-	-	-	-
SyntaxError	-	-	-	-	-	-	-	-	6	4	7	5
TypeError	9	7	7	7	8	5	4	4	7	1	1	1
ValueError	10	8	8	8	9	6	7	6	53	44	41	38
Total Errors	38	30	29	29	55	31	27	24	91	74	55	50

Table 11: Distribution of execution errors for Qwen2.5-Coder-7B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.
[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	10	7	7	3	14	4	5	2	9	7	4	3
FileNotFoundError	1	0	0	0	-	-	-	-	-	-	-	-
ImportError	-	-	-	-	0	1	0	0	-	-	-	-
IndexError	-	-	-	-	2	3	1	1	-	-	-	-
KeyError	2	1	1	1	3	0	0	0	-	-	-	-
KeyboardInterrupt	-	-	-	-	0	1	0	0	1	1	2	2
ModuleNotFoundError	-	-	-	-	1	1	0	0	-	-	-	-
NameError	1	1	0	0	13	8	6	4	10	11	9	10
RecursionError	1	1	1	1	-	-	-	-	-	-	-	-
OSError	-	-	-	-	1	1	1	1	-	-	-	-
RuntimeError	-	-	-	-	1	1	1	0	-	-	-	-
SyntaxError	1	0	0	0	1	1	0	0	4	1	2	1
TypeError	12	5	3	4	8	4	2	1	2	1	2	1
ValueError	12	14	9	9	15	11	9	9	51	38	29	23
Total Errors	40	29	21	18	59	36	25	18	77	59	48	40

Table 12: Distribution of execution errors for Qwen2.5-7B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.
[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AssertionError	1	1	1	1	-	-	-	-	-	-	-	-
AttributeError	14	9	8	8	32	23	21	21	31	13	12	11
FileNotFoundError	2	1	1	1	-	-	-	-	-	-	-	-
IndexError	3	3	3	3	4	4	4	4	-	-	-	-
KeyError	1	1	1	1	3	1	1	1	1	2	1	1
KeyboardInterrupt	1	1	1	3	1	0	0	1	2	2	5	36
NameError	-	-	-	-	6	1	0	0	1	2	1	2
SyntaxError	3	4	3	1	0	1	1	0	13	16	13	3
TypeError	11	10	10	10	7	9	9	9	37	23	25	24
ValueError	14	14	13	13	20	21	20	20	42	56	55	35
Total Errors	50	44	41	41	73	60	56	56	127	114	112	112

Table 13: Distribution of execution errors for Qwen2.5-Coder-3B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	11	9	4	4	29	19	13	11	32	20	13	8
FileNotFoundError	1	0	0	0	6	6	5	5	-	-	-	-
ImportError	1	1	1	1	-	-	-	-	-	-	-	-
IndexError	-	-	-	-	1	1	0	1	0	0	0	1
KeyError	4	3	2	1	4	2	1	1	2	2	2	2
KeyboardInterrupt	2	2	2	3	2	1	2	2	1	1	2	36
NameError	2	1	1	1	2	0	0	1	1	1	3	2
NotImplementedError	1	0	1	0	-	-	-	-	-	-	-	-
RuntimeError	1	1	1	1	1	1	1	1	-	-	-	-
SyntaxError	2	1	1	2	3	3	0	0	14	15	13	3
TypeError	11	8	5	4	10	11	6	4	18	15	11	5
ValueError	9	10	12	10	15	19	18	17	53	58	57	34
Total Errors	45	36	30	27	73	63	46	43	121	112	101	91

Table 14: Distribution of execution errors for Qwen2.5-3B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

C.4 LLaMA Series

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	10	3	1	1	21	6	8	3	27	20	10	9
FileNotFoundError	1	0	0	0	2	3	0	0	-	-	-	-
IndexError	0	1	1	0	2	1	0	0	2	0	3	2
KeyError	1	1	2	2	1	1	2	1	2	2	3	2
KeyboardInterrupt	-	-	-	-	0	4	1	2	-	-	-	-
NameError	1	0	1	0	5	6	5	2	1	2	1	1
RuntimeError	-	-	-	-	3	1	0	0	-	-	-	-
SyntaxError	1	1	1	0	0	1	1	0	19	16	12	7
TypeError	5	5	4	3	7	5	3	2	27	19	13	4
ValueError	14	7	3	4	19	10	7	6	43	40	38	38
Total Errors	33	18	13	10	60	38	27	16	121	99	80	63

Table 15: Distribution of execution errors for Llama-3.1-8B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Error Type	Matplotlib				Seaborn				Plotly			
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
AttributeError	11	7	6	4	28	15	13	11	44	17	11	7
FileNotFoundError	2	0	0	0	4	1	0	0	-	-	-	-
IndexError	2	1	1	2	1	0	0	0	0	0	0	1
KeyError	1	4	1	1	2	3	2	1	1	1	1	1
KeyboardInterrupt	1	1	1	2	1	2	2	2	0	1	1	2
NameError	1	0	0	0	44	9	3	0	4	1	0	0
SyntaxError	3	2	1	0	5	3	3	2	21	24	22	24
TypeError	22	10	10	8	16	12	11	13	47	41	40	36
UFuncTypeError	-	-	-	-	-	-	-	-	1	1	1	1
ValueError	18	16	15	15	20	17	15	15	34	53	57	54
Total Errors	61	41	35	32	121	62	49	44	152	139	133	126

Table 16: Distribution of execution errors for Llama-3.2-3B-Instruct across Matplotlib, Seaborn, and Plotly. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

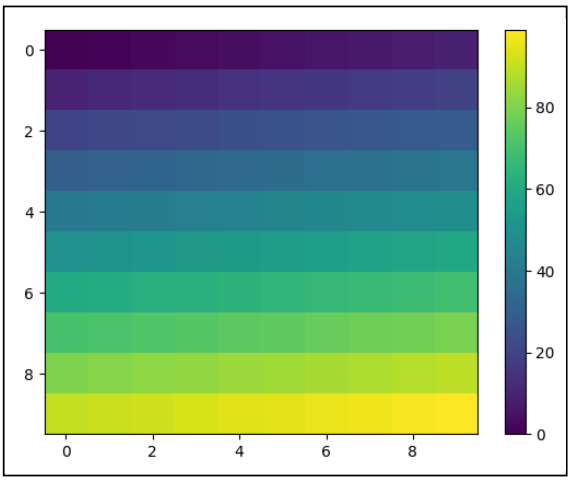
D Case Study

In this section, we present a set of representative examples from VisCoder-7B to illustrate model behavior across the three visualization libraries.

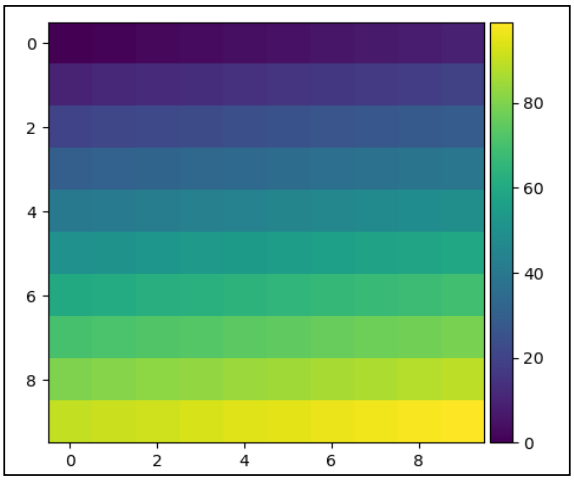
D.1 Matplotlib: Successful Generation

VisCoder-7B: Matplotlib

ID : 11
Task: The code should generate a visual representation of the DataFrame as an image plot. Each cell of the plot corresponds to a value in the DataFrame, with different colors representing different data values. A color bar should be added to the right of the plot to act as a legend, indicating the data value corresponding to each color.
Style: The plot will have a color gradient where colors change according to the values represented. The color bar on the right will help in gauging the data values based on the color shades shown in the plot. Elements like axis visibility, color palette, and layout specifications will be controlled to enhance visual clarity and appeal.



Rendered



Ground Truth

Generated Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Load the dataframe
df = pd.read_csv("dataset/data-11.csv")

# Create the plot
plt.imshow(df, cmap='viridis', aspect='auto')

# Add colorbar
plt.colorbar()

# Show the plot
plt.show()
```

Error: null

Figure 3: Example of a successful generation in **Matplotlib** (ID: 11). The model generates code that executes successfully and produces a plot consistent with the ground truth.

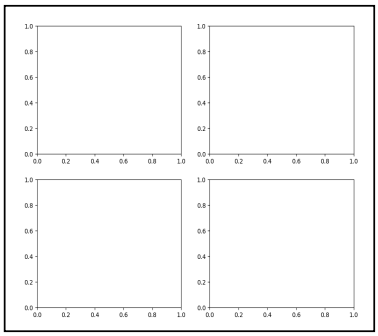
[Back to Appendix Contents]

D.2 Matplotlib: Self-Debug Recovery

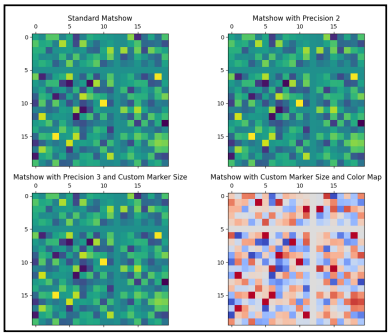
881

VisCoder-7B: Matplotlib

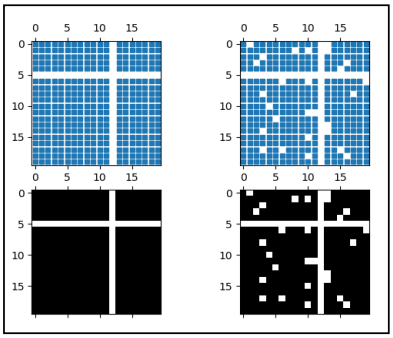
ID : 37
Task: Create a grid of four separate plots (2x2 layout), where each plot will visualize the matrix from the DataFrame. The visualization should reflect the presence or absence of significant values in the matrix, possibly using differing precision and depiction settings across plots to highlight different aspects. Each matrix plot should be customizable in terms of how the data entries are visualized (e.g., adjusting precision or marker size) to allow a nuanced examination of the numerical data.
Style: Apply specific style adjustments to each subplot for clear differentiation. One plot should use standard settings, another with adjusted precision, one with both adjusted precision and marker size, and one more with customized marker size. These variations will aid in better understanding the spread and concentration of significant values in the DataFrame's matrix by altering the granularity and scaling of visual indicators. Adjust axes and tick settings as necessary for optimal clarity and presentation aesthetics. Choose style settings such as color or marker types to enhance legibility and analytical utility.



Nomal Rendered



Debug Rendered



Ground Truth

Normal Generated Code

```
import pandas as pd
import … …
# Load the dataframe
… …
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

axs[0, 0].matshow(df, fignum=False)
axs[0, 0].set_title('Standard Matshow') axs[0,
1].matshow(df.round(2), fignum=False)
axs[0, 1].set_title('Matshow with Precision 2')

axs[1, 0].matshow(df.round(3), fignum=False, cmap='viridis')
axs[1, 0].set_title('Matshow with Precision 3 and Custom Marker
Size')

axs[1, 1].matshow(df, fignum=False, cmap='coolwarm', vmin=-2,
vmax=2)
axs[1, 1].set_title('Matshow with Custom Marker Size and Color
Map')

plt.tight_layout()
plt.show()
```

Self-Debug Generated Code

```
… …
axs[0, 0].matshow(df)
axs[0, 0].set_title('Standard
Matshow')

axs[0, 1].matshow(df.round(2))
axs[0, 1].set_title('Matshow
with Precision 2')

axs[1, 0].matshow(df.round(3),
cmap='viridis')
axs[1, 0].set_title('Matshow
with Precision 3 and Custom
Marker Size')

axs[1, 1].matshow(df,
cmap='coolwarm', vmin=-2,
vmax=2)
axs[1, 1].set_title('Matshow
with Custom Marker Size and
Color Map')
… …
```

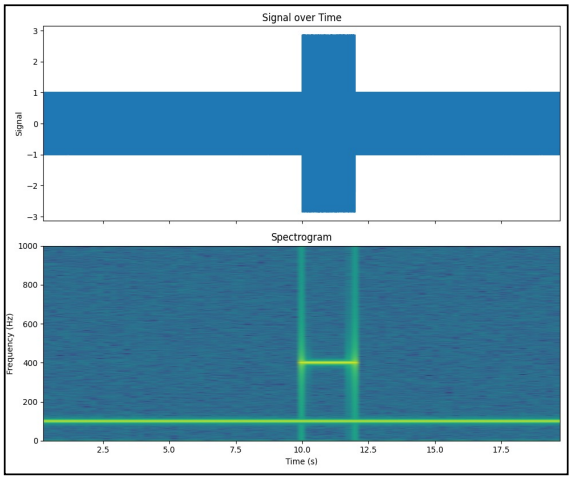
AttributeError: AxesImage.set() got an unexpected keyword argument 'fignum'

Figure 4: Example of a failed generation in **Matplotlib** (ID: 37), where the initial code raises a **AttributeError** and is resolved in the **first** round of self-debug, resulting in a corrected plot that matches the intended semantics.
[Back to Appendix Contents]

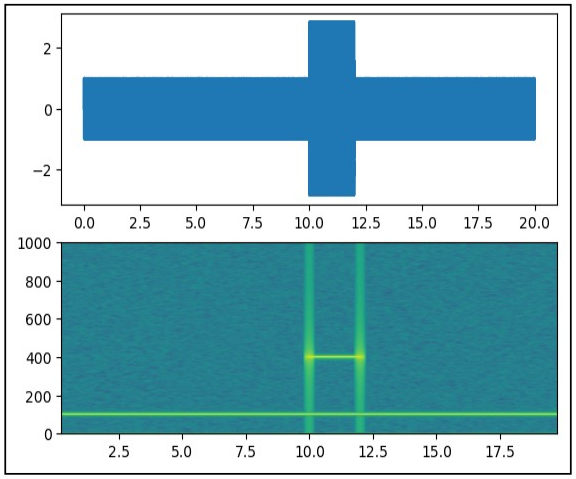
D.3 Seaborn: Successful Generation

VisCoder-7B: Seaborn

ID : 20
Task: The plot should consist of two subplots arranged vertically. The top plot should be a simple line plot illustrating how the 'Signal' varies over 'Time'. The bottom plot should be a spectrogram representing the frequency content of the 'Signal' over 'Time'. The spectrogram plot should use a specific frequency analysis window and overlap parameter.
Style: The top plot should have clear, labelled axes indicating the units of 'Time' and 'Signal'. It should display the full range of the data. The bottom spectrogram plot should include axes indicating time and frequency, with frequency on the vertical axis extending to the adequate range based on the sampling rate used in spectrogram computation. Both plots should share the same horizontal 'Time' axis for easy comparison. Colormaps, axis labels, and layout configurations should enhance clarity and understanding of data representation.



Rendered



Ground Truth

Generated Code:

```
import numpy as np
import pandas as pd
import seaborn as sns
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

# Load the dataframe
pd.read_csv("dataset/data-36.csv")
fig, (ax1, ax2) = plt.subplots(2, 1,
sharex=True, figsize=(10, 8))

# Line plot
sns.lineplot(x='Time', y='Signal',
data=df, ax=ax1)
ax1.set_title('Signal over Time')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Signal')

# Spectrogram
window = signal.windows.tukey(1024)
f, t, Sxx =
signal.spectrogram(df['Signal'],
fs=1/0.0005, window=window, nperseg=1024,
noverlap=512)
im = ax2.imshow(np.log(Sxx),
aspect='auto', extent=[t.min(), t.max(),
f.min(), f.max()],
                origin='lower',
cmap='viridis')
ax2.set_title('Spectrogram')
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Frequency (Hz)')

plt.tight_layout()
plt.show()
```

Error: null

Figure 5: Example of a successful generation in **Seaborn** (ID: 20). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

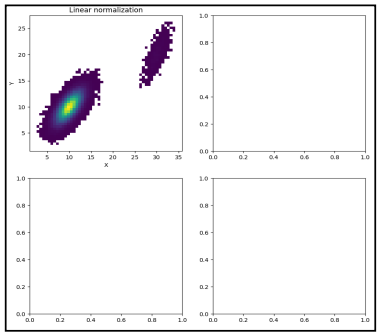
D.4 Seaborn: Self-Debug Recovery

VisCoder-7B: Seaborn

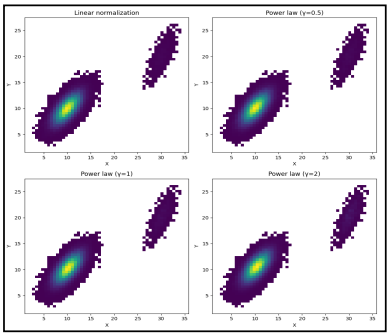
ID : 104

Task: Create a set of four 2D histograms showing the distribution of two variables from the dataframe, arranged in a 2x2 grid format. The first plot should display the data using linear normalization. The subsequent plots should apply a power law normalization with varying gamma values to demonstrate how the distribution visualization changes with different scaling parameters.

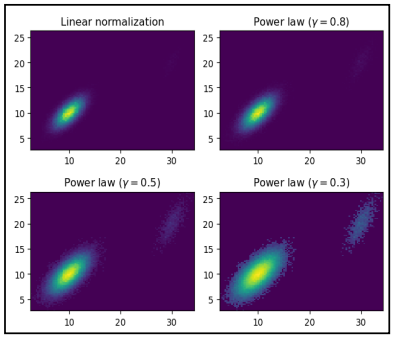
Style: Title each subplot to reflect the type of normalization applied (e.g., "Linear normalization" and "Power law (γ=x)"). Adjust the layout to ensure clear spacing and non-overlapping elements among the subplots. Use a consistent color map across all plots to facilitate visual comparison.



Nomal Rendered



Debug Rendered



Ground Truth

Normal Generated Code

```
import seaborn as sns
from matplotlib.colors import PowerNorm, LogNorm
import … …
df = pd.read_csv("dataset/data-104.csv")

# Create a 2x2 grid of 2D histograms
fig, axes = plt.subplots(2, 2, figsize=(10, 10))

# First plot: Linear normalization
sns.histplot(data=df, x='X', y='Y', ax=axes[0, 0], bins=50,
cmap='viridis')
axes[0, 0].set_title('Linear normalization')

# Second plot: Power law normalization with gamma=0.5
sns.histplot(data=df, x='X', y='Y', ax=axes[0, 1], bins=50,
norm_hist=True, cmap='viridis',
             norm=PowerNorm(gamma=0.5))
axes[0, 1].set_title('Power law (γ=0.5)')

# Third plot: Power law normalization with gamma=1
sns.histplot(data=df, x='X', y='Y', ax=axes[1, 0], bins=50,
norm_hist=True, cmap='viridis',
             norm=PowerNorm(gamma=1))
axes[1, 0].set_title('Power law (γ=1)')

# Fourth plot: Power law normalization with gamma=2
sns.histplot(data=df, x='X', y='Y', ax=axes[1, 1], bins=50,
norm_hist=True, cmap='viridis',
             norm=PowerNorm(gamma=2))
axes[1, 1].set_title('Power law (γ=2)')

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

Self-Debug Generated Code

```
… …
sns.histplot(data=df, x='X',
y='Y', ax=axes[0, 0], bins=50,
cmap='viridis')
axes[0, 0].set_title('Linear
normalization’)

sns.histplot(data=df, x='X',
y='Y', ax=axes[0, 1], bins=50,
cmap='viridis',
             weights=np.power(
df['Y'], 0.5))
axes[0, 1].set_title('Power
law (γ=0.5)’)

sns.histplot(data=df, x='X',
y='Y', ax=axes[1, 0], bins=50,
cmap='viridis',
             weights=np.power(
df['Y'], 1))
axes[1, 0].set_title('Power
law (γ=1)’)

sns.histplot(data=df, x='X',
y='Y', ax=axes[1, 1], bins=50,
cmap='viridis',
             weights=np.power(
df['Y'], 2))
axes[1, 1].set_title('Power
law (γ=2)')
… …
```

AttributeError: QuadMesh.set() got an unexpected keyword argument 'norm_hist'

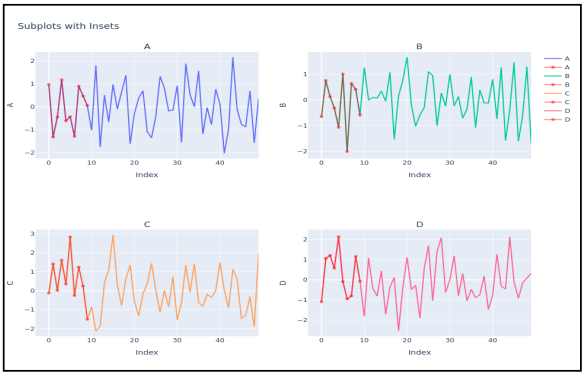
Figure 6: Example of a failed generation in **Seaborn** (ID: 104), where the initial code raises a **AttributeError** and is resolved in the **Third** round of self-debug, resulting in a corrected plot that matches the intended semantics.

[Back to Appendix Contents]

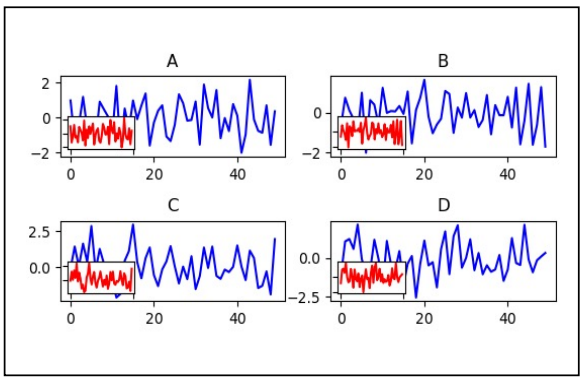
D.5 Plotly: Successful Generation

VisCoder-7B: Plotly

ID : 4
Task: Construct a figure divided into four subplots arranged in a 2x2 grid. Each subplot should display a line plot of one of the dataframe columns against its index. Overlay a smaller inset plot within each main subplot, focusing on a specific area of the line plot, using a contrasting color to distinguish it from the main plot. Each subplot should have its corresponding column label as the title.
Style: For the main plots in each subplot, use a consistent color (e.g., blue) and style for the lines. The insets should additionally feature a contrasting color (e.g., red) for the lines but without tick labels to maintain clarity and focus on the data representation. Adjust the layout to ensure that all elements are well-spaced and visually appealing.



Rendered



Ground Truth

Generated Code:

```python
import pandas as pd
import plotly.graph_objects as go
from plotly.subplots import make_subplots
df = pd.read_csv("dataset/data-4.csv")

# Create the figure with subplots
fig = make_subplots(rows=2, cols=2, subplot_titles=("A", "B", "C", "D"))

# Define the columns to plot
columns = ['A', 'B', 'C', 'D']

# Loop through the columns and create the plots
for i, col in enumerate(columns):
    row = i // 2 + 1
    col_num = i % 2 + 1

    # Main plot
    fig.add_trace(go.Scatter(x=df.index, y=df[col], name=col), row=row, col=col_num)

    # Inset plot
    fig.add_trace(go.Scatter(x=df.index[:10], y=df[col][:10], name=col,
                             line=dict(color='rgba(255, 0, 0, 0.5)')), row=row, col=col_num)
    # Update axes
    fig.update_xaxes(title_text="Index", row=row, col=col_num)
    fig.update_yaxes(title_text=col, row=row, col=col_num)

# Update layout
fig.update_layout(height=800, width=1000, title_text="Subplots with Insets")

# Show the figure
fig.show()
```

Error: null

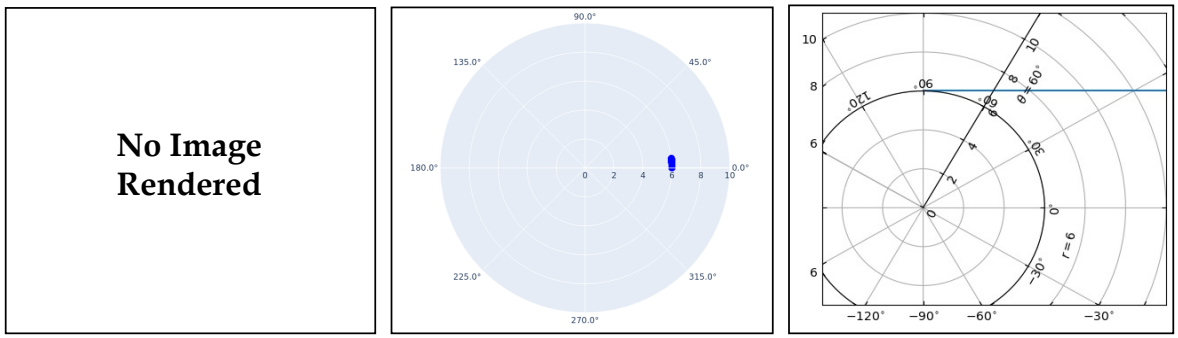
Figure 7: Example of a successful generation in **Plotly** (ID: 4). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

D.6 Plotly: Self-Debug Recovery

VisCoder-7B: Plotly

ID : 15
Task: velop a plot using a polar projection within a rectangular grid. The plot will map the 'Theta' values against 'R' values from the DataFrame. This will require transforming the angular data from degrees to radians and setting up a polar coordinate system in a rectangular plot area. Labels will be added to specific axes to indicate both angular and radial measurements.
Style: The style of the plot should include a grid to enhance readability, with the background consisting of concentric circles and radial lines to represent the angles. Axis labels should be formatted to indicate angles in degrees and radial distances. Display limits for the x and y axes should be explicitly set to frame the plot area properly, ensuring all data points are visible within the defined boundary of the plot. The overall aspect ratio should be adjusted to maintain proportional scaling between axes.



Nomal Rendered | Debug Rendered | Ground Truth

Normal Generated Code

```
import pandas as pd
import plotly.graph_objects as go
df = pd.read_csv("dataset/data-15.csv")

# Convert Theta from degrees to radians
df['Theta'] = df['Theta'] * (3.141592653589793 / 180)

# Create the polar plot
fig = go.Figure(go.Scatterpolar(
    r=df['R'],
    theta=df['Theta'],
    mode='markers',
    marker=dict(size=10, color='blue')
))

# Update layout
fig.update_layout(
    polar=dict(
        radialaxis=dict(visible=True, range=[0, 10]),
        angularaxis=dict(visible=True, tickformat='.1f',
tickformatstop=dict(value='.1f'))
    ),
    showlegend=False,
    width=600,
    height=600,
    margin=dict(l=80, r=80, t=100, b=80)
)
# Show the plot
fig.show()
```

Self-Debug Generated Code

```
… …
# Create the polar plot
fig =
go.Figure(go.Scatterpolar(
    r=df['R'],
    theta=df['Theta'],
    mode='markers',
    marker=dict(size=10,
color='blue')
))

# Update layout
fig.update_layout(
    polar=dict(
        radialaxis=dict(visibl
e=True, range=[0, 10]),
        angularaxis=dict(visib
le=True, tickformat='.1f')
    ),
    showlegend=False,
    width=600,
    height=600,
    margin=dict(l=80, r=80,
t=100, b=80)
)
… …
```

ValueError: Invalid property specified for object of type plotly.graph_objs.layout.polar.AngularAxis: 'tickformatstop'

Figure 8: Example of a failed generation in **Plotly** (ID: 15), where the initial code raises a **ValueError** and is resolved in the **Second** round of self-debug, resulting in a corrected plot that matches the intended semantics.
[Back to Appendix Contents]

E Ethics and Reproducibility Statements

E.1 Potential Risks

This work focuses on training and evaluating large language models to generate Python visualization code. While outputs are validated for executability, there remains a potential risk of generating misleading or malformed plots, particularly in cases of silent failures. No personal, sensitive, or user-generated content is involved.

E.2 Discuss the License for Artifacts

All released artifacts are provided under permissive licenses suitable for academic research. License terms permit use, modification, and redistribution in accordance with each license's conditions.

E.3 Artifact Use Consistent With Intended Use

All external datasets and software components were used in accordance with their original license agreements and intended purposes. Derived artifacts are intended solely for research and educational use, and are not authorized for commercial deployment or redistribution.

E.4 Data Contains Personally Identifying Info or Offensive Content

All data was either synthetically generated or obtained from public sources. Automated filters and manual review were applied to ensure that no samples contain personally identifying information or offensive content. All instructions and tables are free of references to real individuals, groups, or sensitive contexts.

E.5 Documentation of Artifacts

All released artifacts are accompanied by documentation describing their structure, content format, intended use, and evaluation methodology. Sufficient metadata and usage instructions are provided to support inspection, reproduction, and downstream research use.

E.6 Parameters for Packages

All external packages used during training and evaluation were applied in accordance with standard practices. Default parameters were used unless otherwise specified. Any deviations from default settings are documented in the accompanying implementation materials.

E.7 AI Assistants in Research or Writing

Used ChatGPT to capture grammar errors in the manuscript.