Program Augmented LLMs: CS224U Experiment Protocol

Riya Sankhe riyas@stanford.edu
Jenny Yang jjyang1@stanford.edu
Jasmine Bilir jbilir@stanford.edu

1 Introduction

Recent studies have demonstrated that Large Language Models (LLMs) have an impressive ability to decompose a problem into steps and answer a question when prompted with a few input-output exemplars ("few-shot prompting"). In particular, the widely used chain-of-thought (CoT) method presents the model with the explicit intermediate steps that are required to reach the final answer for a particular task. (Wei et al., 2023) Then, the model is expected to apply a similar decomposition to the actual test example, and consecutively reach an accurate final answer (Ling et al., 2017; Amini et al., 2019). However, when CoT prompting is applied to numerical reasoning problems, LLMs often make logical and arithmetic mistakes in their final solution outputs, even when the problem is decomposed correctly (Hendrycks et al., 2021; Madaan and Yazdanbakhsh, 2022).

To tackle this problem, we turn to the novel 'Program of Thoughts' (PoT) prompting method, where LLMs generate executable programs as reasoning steps and then offload the solution execution to a runtime such as a Python Interpreter. In Chen et al, the authors evaluate PoT prompting across 5 math and financial datasets and find that PoT has an average performance gain over CoT of around 12% across all datasets (Wenhu Chen, 2022). Moreover, PoT combined with self-consistency decoding allows their model to achieve SoTA performance on all the math datasets.

Through this project, we aim to tackle two main areas of concern given this context:

- First, Wenhu Chen (2022) uses the OpenAI Codex API (Mark Chen, 2021), a model specifically trained to generate code proficiently.[1] However, since the Wenhu Chen (2022) publication, Codex's successor models are no longer being offered for free. Given the need for OpenAI API private access and the high expenses of such resources, we see that the benefits of PoT will be limited for use-cases where such funding is not available.

- Second, current applications of PoT to numerical reasoning datasets do not comprehensively explore the synergies that can be achieved by modifying and combining different prompting and decoding techniques, including (1) Prompting for self-debugging as explored by (Xinyun Chen, 2023), (2) post-processing to modify slight errors in otherwise executable code, (3) Self-Consistency Decoding.

We hope to address both these concerns by using a low-cost alternative to Codex that has been trained on natural language and not code. We also plan to explore the effects of combining the aforementioned techniques to improve the performance of the model.

2 Hypothesis

We hypothesize that we can produce and improve a low-cost model to serve as a replacement to Codex in the PoT framework. We define a low-cost model to be one that is either publicly available at no cost or one that is available for subscription use at a reduced rate to OpenAI products. While the low-cost model (we ultimately focus on Cohere's Command Nightly model) may not be as proficient as Codex at valid code generation, we plan to use error analysis, self-debugging, self-consistency, and post-processing to directly target issues native to the low-cost model and improve its ability to produce correct results in a PoT framework. Specifically, our goal for improvement will be to target at least one common type of failure with the low-cost model (i.e. Syntax errors, Logic generation errors, etc) and to show that in combination with techniques above we can achieve a greater accuracy

[1] More details on Codex are provided in Section 5.1

and/or qualitative result than the low-cost model alone.

Ultimately, it is our hope that with this guiding hypothesis, we can make PoT prompting more accessible to use-cases where the cost might be prohibitive. We likewise seek to demonstrate that low-cost models can be adapted to meet a minimum standard in a PoT approach.

3 Data

For this project, we focus on two datasets with different levels of difficulty: the GSM8K dataset (Cobbe et al., 2021) and the SVAMP dataset (Patel et al., 2021).

The GSM8K dataset consists of 8.5K high-quality grade school math problems created by human problem writers. These problems take between 2 and 8 steps to solve, and solutions primarily involve performing a sequence of elementary calculations using basic arithmetic operations $(+ - \times \div)$ to reach the final answer. The answer is always a single number. For our purposes, we randomly sample 500 examples from the GSM8K dataset used by (Wenhu Chen, 2022). We use 300 of these as validation examples: to conduct error analysis and evaluate the new prompting techniques we describe in Section 8. We hold out a test set of 200 examples that we will test our final model on, to be able to report the accuracy of our final model as compared to the accuracy of the final model in Wenhu Chen (2022).

The SVAMP (**S**imple **V**ariations on **A**rithmetic **M**ath **W**ord **P**roblems) consists of 1000 arithmetic grade-school word problems. These differ from GSM8K as they generally take 2-3 steps to solve and contain only one unknown, and thus can be thought of as "simpler" than GSM8K. Similarly to the previous dataset, we randomly sample 500 examples from the SVAMP dataset used by Wenhu Chen (2022). We use 300 of these as validation examples and hold out a test-set of 200 examples that we will test our final model on.

Note that our testing and validation datasets are smaller than those used in previous studies due to limited compute resources.

An example of a sample question and answer from both these datasets is given in Table 1.

4 Metrics

On the quantitative side, we will look at the accuracy of the model. This measures the percentage of examples in which the executed code produces the correct numerical answer when compared to the gold output for the math word problem.

On the qualitative side, we aim to look at two metrics: (1) Error breakdown and (2) Code quality.

For error breakdown, of the questions that our model is unable to answer correctly, we will quantitatively break down the percentage of examples which fall into error categories as defined in Section 7.3. Throughout our model improvement process, we would like to iteratively reduce the rates at which these errors occur. We also plan to break down our model's accuracy across different types of questions. For code quality, we plan to perform a qualitative assessment of the code quality produced by our model. We will be looking for signs of high-quality code, such as good variable names, no useless or unused variables, and clear problem decomposition.

5 Large Language Models

We focus on two different LLMs to compare the performance of PoT prompting on our two datasets.

5.1 LLM 1: Codex

We compare our results to few-shot prompting applied using the OpenAI Codex (code-davinci-002) API. This model was trained on both natural language and billions of lines of source code from publicly available sources, including code in public GitHub repositories. It has 175B parameters. Mark Chen (2021) finds it is specifically trained to parse natural language and produce executable code, and it has been successfully applied for code generation, refactoring code and transpilation (the process of translating one programming language to another). We aim to compare the performance of our low-cost model against the performance of this specially trained model and see how our methods of self-debugging, post-processing, and adapted prompting can help improve performance to the level of this model.

5.2 LLM 2: Command Nightly

We then evaluate PoT prompting applied to Cohere's generative Command Nightly (command-xlarge-nightly) API. The Cohere model has 52.4B parameters (3x smaller than Codex) and is majority trained on natural language. While there is not much further information on Cohere's pre-training,

Table 1: Examples of Input and Output Format from GSM8K and SVAMP Datasets

Dataset	Input	Gold Output	Process

GSM8K	Josh decides to try flipping a house. He buys a house for $80,000 and then puts in $50,000 in repairs. This increased the value of the house by 150%. How much profit did he make?	$70,000	`cost_of_original_house = 80,000`
			`increase_rate = 1.5`
			`value_of_house = (1 + 1.5) * 80000`
			`cost_of_repair = 50000`
			`ans = 200,000 - 50,000 - 80,000`

SVAMP	Each pack of dvds costs 76 dollars. If there is a discount of 25 dollars on each pack, how much do you have to pay to buy each pack?	$51	`ans = 76 - 25`

Cohere claims it is mainly adapted to instruction-like prompts.

6 General Reasoning

During our research we investigated the GSM8K dataset and the SVAMP dataset. As described above, the examples in these datasets take the form of math word problems (MWPs) that take on the form of a natural-language narrative and a question requiring a numerical answer. Both these datasets were instrumental in Wenhu Chen (2022), where it was a necessary criteria to improve model performance on prompts mixing natural language and numeric information. For this reason, we believe the information and understanding they provide will be beneficial to achieving our hypothesis as well offer direct comparison to previous PoT work.

Additionally, we believe it is possible to analyze the various errors that occur and target them with specific mediating tactics. In (Xinyun Chen, 2023), naive techniques such as simple feedback re-prompting showed improvement of 8% on code generation from text-trained models. Likewise, in Wenhu Chen (2022), using methods of self-consistency in few-shot prompting for PoT generally showed improved performance over standard PoT. Because these methods have been shown to improve models with similar goals, we believe we can reduce the number of naive errors made by our low-cost model using a combination of such techniques. Specifically, these additions can be developed to make the low-cost PoT solution more comparable to the behavior of Codex PoT without additional training or cost.

Particularly, our baseline(s) show that Cohere Command Nightly can produce executable programs (See Summary of Progress: Cohere Baseline Experiments). However, we notice significant errors of the types described fully in our Summary of Progress. Based on our understanding of each error type, we plan to address them using case-by-case tactics and experiments. For example, we plan to address errors where problem approach was correct but syntax was incorrect using post-processing techniques that can make small alterations, rendering the program correct and executable. For specifics on each error type and intended approach, please see Future Work.

Ultimately, we reason that using these reputable tactics and industry-standard datasets, we can show improvement of our Cohere baseline and make way towards proving our hypothesis. Given our experiments, we will also likely have a greater understanding of Codex PoT's unique strengths and will be able to use both outputs from Wenhu Chen (2022) and our work to draw valuable conclusions about PoT for various LLMs.

7 Baseline Experiments

7.1 Huggingface Baseline Experiments

In the initial exploration for this project, we explored several different types of low-cost mod-

els. We first explored publicly accessible code-generation models available for free on hugging face. Primarily, we explored different size variants of the CodeParrot and the Salesforce CodeGen models. We faced several barriers with these models.

1. These models could not be prompted with the full-set of prompts and exemplars as used in Wenhu Chen (2022). We encountered issues with prompt truncation and token size constraints.

2. Additionally, these models are trained with multiple use-cases in mind, including code summary and code generalization. Because of their multi-purposes, these models performed poorly in understanding that prompts requested executable solutions.

3. We tried variations of these models (ranging from small 110M parameters models to large 1.5B+ parameter models) and found that degree of improvement did not change greatly with model-size.

7.2 Cohere Baseline Experiments

For our Cohere model, we used the base prompt from Wenhu Chen (2022) which implements few-shot prompting in the form of six question-code pairs with leading text indicators to show whether what follows is a question or code solution. For each question, we appended to base prompt the leading text indicator for a question, the question, and the leading text indicator for a code solution.

The results for our baseline with the Cohere Command Nightly Model and the Codex Baseline are summarized in Table 2.

7.3 Error Analysis

We then analyzed the errors produced by our model on the two datasets. We aimed to understand these errors so we could identify common failure modes of PoT prompting applied to our LLM, and thus apply targeted techniques to improve the accuracy of our model.

- **Syntax Errors:** This error indicates that the model produced code with syntactical errors. There were three main types of syntax errors:(1) When the model incorrectly references a previous variable (2) when the model does not store the final result in an 'ans' variable, and (3) when two computations and equalities are represented in one line of code. These result in code that is not executable and thus throws an error. We name these Syntax Errors.

- **Logic Generation Errors:** This error indicates that while the code did execute, it did not reach the right answer. As a result, there is a bug in the logical progression of code. We name these Logic Gen Errors.

- **Natural Language Output Errors:** This error indicates that the model generated an output in natural language instead of code, decomposing the problem into steps in a CoT style as opposed to generating Python code. We name these NL Output Errors.

- **No Problem Decomposition Errors:** This error indicates that the model did not produce any steps before directly printing a numerical answer. These answers were often incorrect, with no visibility into the steps the LLM took to get to its output. We name these NPD Errors.

An overall breakdown of the percentage of examples from our validation set that conforms to each of these errors can be found in Table 3.

It is noteworthy that 49.6% of the errors on GSM8K were caused by Syntax Errors. However, within these 124 syntax errors on the 300 sample validation set, 54 were causing by a missing 'ans' variable and 70 were due other syntax errors.

8 For the rest of the quarter...

In the remaining weeks, we plan to implement various extensions to boost our baseline performance. The three main extensions we will implement to tackle our current four broad categories of errors are:

- **Post-processing:** This will help catch common syntax errors and clean up the code so that it can be executed successfully. The post-processing will mainly focus on correcting incorrect variable names, renaming the variable in the final line to an 'ans' variable, and stripping leading natural language text.

- **Self-debugging:** Following the work by Xinyun Chen (2023) we will implement self-debugging to help the model edit its own

Figure 1: Error Types and Example Cases

Type 1 Error: Syntax Errors

Context	The expenditure of Joseph in May was $500. In June, his expenditure was $60 less.
Question	How much was his total expenditure for those two months?
Output	'May_expenditure = 500 June_expenditure = 500 - 6 total_expenditure = May_expenditure + June_expenditure' 'May_expenditure = 500 June_expenditure = 500 - 6 total_expenditure = May_expenditure + June_expenditure ans=total_expenditure'

Type 2 Error: Logic Generation Errors

Context	After transferring to a new school, Amy made 20 more friends than Lily.
Question	If Lily made 50 friends, how many friends do Lily and Amy have together?
Output	'num_of_friends_that_amy_made = 20 num_of_friends_that_lily_made = 50 total_friends = num_of_friends_that_amy_made + num_of_friends_that_lily_made ans = total_friends' 'num_of_friends_that_lily_made = 50 num_of_friends_that_amy_made = num_of_friends_that_lily_made + 20 total_friends = num_of_friends_that_amy_made + num_of_friends_that_lily_made ans = total_friends'

Type 3 Error: Natural Language Output Errors

Context	Twenty dozen cups cost $1200 less than the total cost of half a dozen plates sold at $6000 each.
Question	Twenty dozen cups cost $1200 less than the total cost of half a dozen plates sold at $6000 each.
Output	'The total cost of buying each cup is $1200 / 20 = $60.' 'cost_half_dozen=6*6000 cost_twenty_dozen=cost_half_dozen-1200 cost_one = cost_twenty_dozen/(20*12) ans = cost_one'

Type 4 Error: No Problem Decomposition Errors

Context	Baldur gets water from a well. He gets 5 pails of water every morning and 6 pails of water every afternoon.
Question	If each pail contains 5 liters of water, how many liters of water does he get every day?
Output	'ans = 5 * 5 + 6 * 5 = 25' 'morning_pails=5 afternoon_pails=6 total_liters = 5*(morning_pails+afternoon_pails) ans=total_liters'

Table 2: Few-shot results for Codex Model (Baseline 1) and Command Nightly Model (Baseline 2)

Dataset	Baseline 1 Accuracy	Baseline 2 Accuracy
GSM8K	71.6%	16.7%
SVAMP	85.2%	65.0%

buggy code. For syntax bugs, we plan to use this approach in parallel with and as a comparison to manual post-processing to identify the strengths and weaknesses of each approach. For logical errors, we hope to provide examples of self-debugging in our prompt to teach the model what bugs may look like.

- **Self-consistency:** To make our model more robust, we plan to implement self-consistency by extracting multiple answers from the model and executing them each. We will then choose the most common answer as the final answer.

While we will be focusing our testing and experimentating on the SVAMP dataset, we hope that our framework will boost accuracy on the GSM8K dataset as well.

8.1 Obstacles

The biggest obstacle that we foresee is working with the Cohere model with a trial API key. While we can handle the slower processing rate, towards the end of the project we may run out of the free trial credit. However, if this occurs we plan to make other free accounts or to pay a small amount.

Another obstacle is the effectiveness of techniques like self-debugging. These techniques have not been widely applied in numerical reasoning contexts, and they are often more successful when you can provide the model information about the incorrectness of the code, including code execution output or unit test output. We wonder what the

Table 3: Percentage of Examples that fall into each error category

Dataset	Syntax Errors (%)	Logic Gen Errors (%)	NL Output Errors (%)	NPD Errors (%)
GSM8K	49.6%	29.6%	10.4%	10.4%
SVAMP	20.0%	80.0%	0.0%	0.0%

effectiveness of simple feedback (for example just saying, "This code is wrong, fix it") as explored in Xinyun Chen (2023) will be, and whether this will actually lead the model to performance improvement.

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics.

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 158–167, Vancouver, Canada. Association for Computational Linguistics.

Aman Madaan and Amir Yazdanbakhsh. 2022. Text and patterns: For effective chain of thought, it takes two to tango.

et. al. Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2211.12588*.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2080–2094, Online. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models.

Xinyi Wang William W. Cohen Wenhu Chen, Xueguang Ma. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Nathanael Schärli Denny Zhou Xinyun Chen, Maxwell Lin. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.