How Much is Lost When We Cut the Cost of Program of Thoughts?

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

Abstract

Recently, there has been significant research into prompting LLMs to generate step-by-step reasoning in the form of programs to solve complex tasks. Program-of-Thoughts prompting is a state of the art method that first uses LLMs to generate programming language statements, and then offload the computation to a program interpreter. However, a majority of PoT research uses the LLM Codex, which is expensive and inaccessible. Through this project, we produce and improve a low-cost alternative to Codex to solve math word problems: Cohere's Command Nightly model. We implement four different methods to improve the accuracy of the model: (1) Prompt Manipulation, (2) Self-Consistency, (3) Post-processing of Code Outputs for syntax errrors, and (4) Self-debugging for syntax errors. We find that self-consistency decoding creates the biggest performance improvement in our pipeline, and other techniques marginally improve accuracy over the baseline. Overall, the performance still falls short of the Codex model, and thus our findings suggest that LLMs such as Codex that are trained on code improve the performance of PoT prompting, partly justifying their expense.

1 Introduction

Recent studies demonstrated that Large Language Models (LLMs) are able 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 a 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 Wenhu Chen (2022), 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:

1. 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 expense of such resources, we see that PoT will be limited for use-cases where funding is not available.

2. Current applications of PoT to numerical reasoning datasets do not comprehensively explore the modification and combination of different prompting and decoding techniques, including (1) Prompt Manipulation, (2) Self-Consistency Decoding, (3) Prompting for Self-Debugging as explored by (Xinyun Chen, 2023), and (4) post-processing to modify slight errors in otherwise executable code.

Through this project, we aim to 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

[1] More details on Codex are provided in Section 4.1

rate to OpenAI products, focusing on Cohere's Command Nightly model in this project. We show that through error analysis, prompt manipulation, self-consistency, generated code post-processing and self-debugging, we are able to improve the performance of a baseline model, but are not able to reach the same accuracy as the Codex model. We conclude that while these techniques are marginally helpful in improving performance, using an LLM trained on code vastly improves the performance of PoT prompting, providing some justification for the expense of models such as Codex.

2 Prior Literature

Code generation to solve quantitative problems originated first as a extension of chain of thought prompting. Wei et al. (2023) first introduce the idea of encouraging models to use intermediate reasoning steps in a chain of thought format, allowing complex reasoning abilities of large language models to be significantly improved. More formally, they demonstrate that this behavior emerges without fine-tuning in LLMs when prompted using input-output exemplars. There are four main benefits of using chain-of-thought prompting to facilitate reasoning: 1) it allows models to decompose multi-step problems, 2) it provides an interpretable window into the behavior of the model to reveal how it arrived at an answer, 3) it can be used for tasks like math word problems, commonsense reasoning, and symbolic manipulation, with the potential to be applicable to any tasks that are solved via language, 4) it can be applied to any sufficiently large language models simply with examples of chain of thought sequences. The authors demonstrate that using chain-of-thought prompting improves performance on a range of arithmetic, commonsense, and symbolic reasoning tasks via experimentation on 12 datasets.

Exploiting the power of prompts in affecting LLM output, Shizhe Diao (2023) propose "active CoT prompting" where the model generates its own reasoning annotations with respect to a query, and then scores itself based on uncertainty. For cases where uncertainty is high, it seeks additional human annotation, greatly reducing the human interaction required. Using datasets typical to chain-of-thought evaluation, including commonsense resoning, arithmetic reasoning, symbolic reasoning, etc., this model outperformed standard chain-of-thought and several other chain-of-thought variants, including random and auto CoT. More specifically in airthmetic reasoning, "compared with the competitive baseline, self-consistency, Active-Prompt (D) outperforms it by an average of 2.1% with code-davinci-002. Larger improvement is observed with text-davinci-002, where ActivePrompt (D) improves over self-consistency by 7.2%" measured in answer accuracy (Shizhe Diao, 2023). Overall, it appears that when used in tandem with large industry-accepted LLMs, active-prompting can show improvements in numerical questions.

Rather than changing the prompt to reason better in natural language, Wenhu Chen (2022) devise a method of coaxing the model to express reasoning via executable Python programs and then delegating the actual computational task to an external Python interpreter. In this work, the authors propose the Program of Thoughts (PoT) framework for solving numerical reasoning problems. Using PoT, LLMs are only responsible for expressing the problem reasoning process in the programming language, not the final answer. Moreover, PoT is also significantly different from direct code generation. First, it stimulates models to break down equations into a multi-step thought process, and second, it binds semantic meanings to variables to ground the model in language.

The authors evaluated PoT across five math word problem datasets (MWP) and three financial datasets. Under both few-shot and zero-shot settings, PoT significantly outperformed CoT, and direct code generation across all the evaluated datasets. This was the motivation for us to carry out a similar project.

Similarly to Wenhu Chen (2022), Luyu Gao (2021) use an LLM to read natural language problems, generate programs as the intermediate reasoning steps, and execute the Python code to obtain the solution in "Program-Aided Language Model (PAL)." The only learning task for the LLM is to decompose the natural language problem into runnable steps. Specifically, the LLM generates both its chain-of-thoughts in natural language as well as augmenting each natural language step with its corresponding programmatic statement. This teaches the model to generate a program that will provide the answer for a given question rather than relying on the LLM to perform the calculation correctly. The natural language intermediate steps are generated with comment syntax (#) so that the entire output can be fed directly into the interpreter. They validate the performance of PAL on mathematical, symbolic, and algorithmic reasoning tasks, comparing the performance of PAL with direct prompting and chain of thought prompting. The authors found that across all tasks, PAL with GPT-3 Codex, a code generation

language model, achieves few-shot state of the art performance. The jump in performance with PAL over CoT remains consistent even with weaker LMs.

Taking a step further from using LLMs for just code generation, Xinyun Chen (2023) propose "self debugging" as a way to teach language models to debug their predicted program code via few-shot demonstrations in “Teaching Large Language Models to Self Debug." By leveraging feedback messages and reusing failed predictions, this method improves sample efficiency and achieves state-of-the-art performance on the Spider dataset for text-to-SQL generation, TransCoder for C++-to-Python translation, and MBPP for text-to-Python generation. The authors use three types of feedback messages that can be automatically acquired and generated with code execution and few shot prompting. First, they explore self-debugging with simple feedback where the feedback is a sentence that indicates the code correctness without more detailed information. Second, they explore self debugging with unit tests, where the model is also presented with the execution results of unit tests in the feedback message. Lastly, they explore self-debugging via code explanations, where the model has to self-debug by explaining the generated code. On text-to-SQL generation where there is no unit test specified for the task, leveraging code explanation for self-debugging consistently improves the baseline by 2 - 3%, and provides a performance gain of 9% on the hardest problems. For code translation and text-to-Python generation tasks where unit tests are available, self-debugging significantly increases the baseline accuracy by up to 12%.

3 Data

We focus on the SVAMP (**S**imple **V**ariations on **A**rithmetic **M**ath **W**ord **P**roblems) dataset for the project (Patel et al., 2021) This dataset consists of 1000 high-quality grade school math word problems. They generally take 2-3 steps to solve and focus on one of the basic arithmetic operations $(+ - \times \div)$ to reach the final answer, which is always a number. For our purposes, we randomly sample 600 examples from the SVAMP 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 5. We also use a test set of 300 examples that we 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). 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 the dataset is given in Table 1.

4 Large Language Model Overview

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

4.1 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 prompt manipulation, self-debugging, post-processing, and self-consistency can help improve performance to the level of this model.

4.2 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 primarily trained on natural language. While there is not much further information on Cohere’s pre-training, Cohere claims it is mainly adapted to instruction-like prompts.

5 Methods

5.1 Huggingface Baseline Model

We initially explored several different low-cost models. We first experimented with publicly accessible code-generation models available from Hugging Face. Primarily, we explored different size variants of the CodeParrot and the Salesforce CodeGen models. However, we face several barriers:

1. These models could not be prompted with the full length prompts as used in Wenhu Chen (2022), since there were 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

Table 1: Examples of Input and Output Format from SVAMP Dataset

Input	Gold Output	Process
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`

multi-purposes, these models performed poorly in understanding that prompts requested executable solutions.

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

5.2 Cohere Baseline Model

We thus turn to a Cohere Baseline model. Our baseline model prompts the LLM with 5 question-code pairs as done in Wenhu Chen (2022). To elicit the LLM's ability to generate code, we use the text "Read the following passages to answer questions with Python code, store the result as a 'ans' variable." For each question, we append the question to this base prompt and record the generated code. We provide the full few-shot prompts in Appendix A.

5.3 Prompt Manipulation (PM)

To address logic generation errors, we customize the sample problems in the prompt for each math problem. To do this, we first classify each math problem as either addition, subtraction, multiplication, or division. This is done with few-shot prompting with the same Cohere model to find the type of question. The full prompt for question type classification is shared in Appendix A.2.

Once the model predicts a question type, the prompt for code generation is customized based on the question type. The base prompt contains two questions: a two step addition and subtraction problem and a problem with no math required due to the answer being in the problem. Based on the question type, three to four examples of in the same question type are appended to the prompt.

5.4 Self-Consistency (SC)

We leverage self-consistency (SC) decoding to replace the naive greedy decoding of our pipeline. We set a temperature of 0.9 and $K = 5$ throughout our experiments. We then set our prediction to the most consistent answer from the 5 code generations. We create one model that uses SC with PM (named "PM+SC") and one model that uses SC with PM, self-debugging and post-processing (named "SC+PM+SD+PP")

5.5 Prompting for Self-Debugging (SD)

We evaluate prompting for self-debugging on our two main types of errors: code-based syntax errors and logic generation errors (See Section 6.1).

For syntax errors, we design few-shot prompts that re-use failed predictions with syntax errors and crafted feedback messages to teach the LLM to debug a given program. The feedback messages contain the error messages outputted by the program and instructions on how to correct the error. An example of this is shown in Appendix A.

For logic generation, we design few-shot prompts that re-use failed predictions and re-prompt the LLM to correct the code if it is incorrect. However, we find this is ineffective in correcting the code (discussed in 6) and thus we do not include this in our final model.

5.6 Code Post-Processing (PP)

We used post-processing techniques to address issues of unexecutable code due to syntax errors. In particular, we worked to identify and target repetitive syntax errors of three specific types:

1. Errors where the model produced code with a natural language preface: Ex. "Using Python\n[Executable Code]"

2. The model produced code with unexpected indentations (leading tabs and spaces on new-lines) : Ex. " var=1\n var=2 ..."

3. The model produced code with un-executable simplifications : Ex. "var=1\nvar2 = 1 - var1 = 0 \nans=var2"

Because these issues occurred the most frequently and were highly generalized, we used regular expressions (regex) to identify and reprocess these outputs, and then attempt re-execution. For more information on our results, see Section 6.

Other types of common syntax errors proved more difficult to capture with brute-force regex expressions and string methods due to the subtleties and diverse responses of code generation. Thus, even

though similar syntactical errors occurred, parsing out variables names and ensuring maintenance of all other existing and functioning code proved to require highly specialized case-by-case handling. We felt that problem specific post-processing did not yield further information and would likely prove useless for other datasets. For more information on this decision, see our project limitations in Section 7.

5.7 Final Pipeline

In order to harness the power of all our resources we developed a final pipeline that utilized the strengths of the aforementioned methods. Overall, this pipeline builds off of the baseline cohere-nightly model with the following steps:

1. Using the LLM, determine what type of math question the input is
2. Perform prompt modification based on the question type
3. Prompt the LLM with temperature 0.9 to generate 5 code outputs (for self-consistency)
4. Use self-debugging and post-processing on the outputs with syntax errors to make it executable
5. Execute all code outputs
6. Select the most common answer as the output

6 Results and Analysis

6.1 Cohere Baseline

We give our results in Table 2. The model is able to achieve an accuracy of 65% on our validation set as compared to the accuracy of 85.2% for the same prompting technique with Codex.

We conduct an error analysis to identify common failure models of PoT prompting so we can apply targeted techniques to improve the accuracy of our model. We break down errors into two main categories:

- **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. 7% of dataset examples have 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. 28% of examples have logic generation errors with our baseline model.

6.2 Prompt Manipulation

To try to reduce logic generation errors, we implement prompt manipulation to customize prompts based on question type. The first stage, predicting what type of question it is, has 61% accuracy when compared with the SVAMP provided labels.

Prompt manipulation based on the predicted labels reduces the overall accuracy of the model to 62% on our validation set. Out of the correctly predicted problem types (183 total), the overall accuracy was 64.5% and on incorrectly predicted problem types (117 total), the overall accuracy was 58.1%. This shows that prompting with samples of the same question type as the actual problem improved accuracy, and a main bottleneck on this method was simply the ability to correctly predict the question type. However, this could be due to the confounding variable of question difficulty, where it may be easier to classify one step problems correctly and thus their accuracy is higher too.

Across all samples, prompt manipulation also reduces the overall number of logic generation errors. At the same time, it introduces more syntax errors. We theorize that perhaps having less diverse examples in the prompt worsened the overall syntax, especially in cases where the samples were not relevant to the problem type.

6.3 Self-Consistency

To reduce the number of logic generation errors, we leverage self-consistency decoding as described in Section 5.4. According to Table 2, we found that PM+SC outperforms PM by 7% on the SVAMP dataset, which provides evidence for the intuition that numerical reasoning problems can admit multiple different ways of thinking leading to a unique correct answer. (Wang et al., 2023).

6.4 Post-Processing on Non-Executable Self-Consistency Examples

On the 300 case validations set, the process of creating 5-shot examples generated 1290 non-executable code strings. The method of post-processing identified 204 of these as falling in the generalized error categories from 5.3 and 5.2. Post-processing was able to make fix 133 of these code strings and make

Figure 1: Final model pipeline (see Section 5.7 for details)

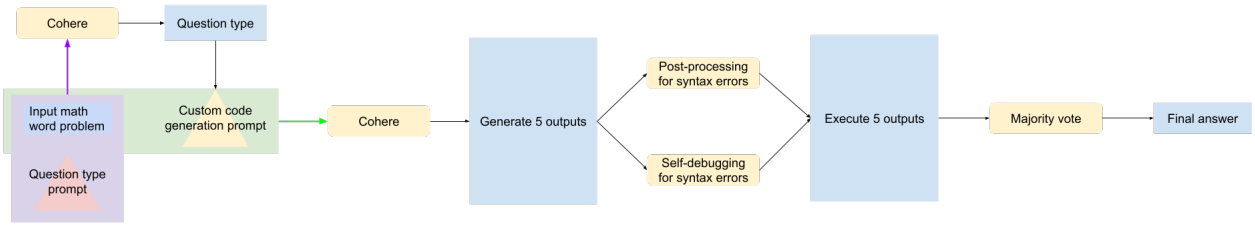

Table 2: Model Performance on Validation Set: "PM" represents the baseline with Prompt Manipulation, "PM+SC" represents the baseline with prompt manipulation and self-consistency, "PM+SC+SD+PP" represents the baseline with prompt manipulation, self-consistency, self-debugging and code post-processing

Model	Accuracy	Logic Generation Errors	Syntax Errors
Codex Baseline	85.3%	N/A	N/A
Cohere Baseline	65%	28%	7%
PM	62%	27%	11%
PM + SC	69%	N/A	N/A
PM + SC + SD + PP	67.67%	N/A	N/A

Figure 2: 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'

them executable. This showed an improvement on SC example execution of 10.31%. Of these, the majority (110) were resolved using a single invalid term clean method of post-processing. Of the 71 strings that were not reliably fixed by our methods, 27 were entirely natural language responses where the model had failed to be produce code responses. The post-processing method of resolving unnecessary simplifications was identified 25 times and was resolved on 18 instances. The post-processing method of unexpected indents fixed 3 out of 13 identified errors.

In examining logs from SC example executions, we were able to better understand that primary gains in performance attributed to post-processing occurred from the simple case of removing invalid terms and natural language prefaces made by the baseline model. These cases were well-suited to post-processing because they were a common error of the baseline model. We speculate this is true because of the baseline model's natural language foundations. Given that codex is trained to produce code, the authors of Wenhu Chen (2022) did not have these errors in the first place.

The goals in the development during post-processing was to recover as many code pieces as possible. However, in section 6.5 we will discuss how answers that require post-processing may have decreased overall pipeline results. We speculate that code recovery may not always be beneficial.

6.5 Self-Debugging

We try self-debugging separately for our two types of errors: syntax errors and logic generation errors. We make a few interesting observations for self-debugging for syntax errors:

- Unlike demonstrated in Xinyun Chen (2023), self-debugging with simple feedback and no other information("This code is wrong, fix it") was not effective in improving our model's performance. In fact, our experiments with re-prompting the model in this way often resulted in code with even more syntactical and logic generation errors than before. We hypothesize that for a model like Command Nightly that is

primarily trained on natural language, it may be comparatively more difficult to identify syntax errors in code without specific instructions: such a task might be easier for a model like Codex as used in Xinyun Chen (2023)

- We first try few-shot prompting with examples where the model makes a specific syntax error (for example mis-references a previously declared variable), is given feedback in the form of an error message and "This statement is wrong" and then corrects the code.(An example of such a prompt is provided in Appendix A.3) We find this is less effective than giving the model very specific instructions about how to correct the code. For example, for the error where the model mis-references a variable, telling the model a variable is misspelled and the line it is on is more effective in getting it to correct the code. An example of such a prompt is provided in Appendix A.4

Overall, while self-debugging for syntax errors does help when we provide very specific instructions, we conclude that it is not scalable, as it would require granular knowledge about the types of errors the model was making in advance. Similarly, we find that self-debugging with simple feedback or few-shot prompting also does not help improve accuracy for logic generation errors. Thus, in our final model, we only implement self-debugging for one specific types of syntax errors: errors where the model mis-references a previously declared variable. The results on this are discussed in the next section.

6.6 Post-Processing and Self-Debugging on Final Pipeline

We find that our pipeline with post-processing and self-debugging after prompt manipulation and self-consistency(PM + SC + SD + PP) does not outperform the model with just prompt manipulation and self-consistency (PM + SC).

In examining our logs from these methods, we see that PP+SD accomplish the goal of making a larger number of SC examples executable. In the vanilla SC approach, on average 3 out of 5 examples were executable. With full post-processing and debugging, on average 4 out of 5 examples were executable. Out of 300 validation problems, SC and the final pipelines produced different predictions on 60 of the cases. 28 of these cases accounted for times when post-processing and self-debugging improved the model, predicting the correct answer where SC alone could not. On the remaining 32 examples, SC had predicted the correct answer and the final pipeline (though getting more SC code pieces to execute) did not get the correct answer. We noticed that in the 32 cases, post-processing revealed additional answers that led the model astray in making the correct final decision. Take for example, this problem where the correct answer was 30. The SC-only model produced only 3 executable code pieces which produced [30, 27, 30] as answers to the problem. SC with post-processing and self-debugging made all 5 examples executable, generating the following answers [30, 27, 30, 27, 27]. Accordingly, upon final prediction, these incorrect answers swayed a final prediction 27, making the overall prediction incorrect. This in part may indicate that cases where post-processing is required might already indicate that the model is on a tangent which will not produce a correct answer.

There were mixed results in terms of correlation with any specific post-processing error and incorrect answers. Removing invalid terms accounted for 18 out of the 32 incorrect answers that differed from SC-alone, but also accounted for 11 of the 28 improved answers over SC-alone. Likewise, removing unnecessary simplifications and unintended indents were responsible for a roughly even number of improvements and failures compared to SC. Self-debugging, however, was used in a total of 7 of the 60 cases where SC and the full-pipeline differed and improved the SC each time. Thus, we believe that spelling constituted a strong and valid use case in which self-debugging showed to be a strong supportive method in numerical reasoning.

6.7 Performance on Test Set

To validate the generalisability of our final pipeline, we ran it on 300 samples from SVAMP that did not appear in the validation set or as samples in any prompts. The accuracy on our test set was 61%, in line with the accuracy we saw on the validation set.

7 Conclusion

Through this project, we produce and improve a Cohere's Command Nightly model to serve as a replacement to Codex in the PoT framework when applied to math word problems. This model was chosen due to its cheaper cost and training data of both natural language. We implement four different methods to improve the accuracy of the model on 300 randomly chosen examples from the SVAMP dataset: (1) Prompt Manipulation, (2) Self-Consistency, (3) Post-processing of Code Outputs for syntax errrors,

and (4) Self-debugging for correcting misspellings in references to previously declared variables. We find that self-consistency decoding creates the biggest performance improvement in our pipeline, increasing accuracy from 62% to 69%. Post-processing helps make more code outputs executable, but ultimately decreases the accuracy of the model, probably because code outputs with some syntax errors might also contain other errors. Self-debugging helps make more code outputs executable and improves the accuracy of the model, but has limited use cases as it can only be used when we have granular understanding of the types of errors we will see. Overall, the performance still falls short of the Codex model. Thus, while these techniques are marginally helpful in improving performance, our findings provide evidence to suggest that an LLM trained on code improves performance of PoT prompting, providing some justification for the expense of models such as Codex.

Known Project Limitations

- The SVAMP dataset (like many NLP datasets) contains examples that are flawed or labeled incorrectly. Take for instance the example SVAMP question "You had 14 bags with equal number of cookies. If you had 28 cookies and 86 candies in total, how many bags of cookies do you have?" with the given answer 2. The problem is phrased oddly, and the correct answer is given the prompt. There are 14 bags of cookies, not 2. We acknowledge that this would cause issues for accuracy and penalize the model when actually correct; however, going off of SVAMP as an industry-grade dataset and directly comparable to Wenhu Chen (2022), we have chosen to utilize it despite limitations.

- Certain methods we use such as post-processing are highly specialized to the dataset. One would need to see if the types of syntax errors made by the model are generalizable across datasets to see if these post-processing techniques would helpful; however, we were intentional about choosing post-processing and self-debugging techniques that were not overfitted to any one specific type of question.

- It is our intention in future work to take a larger test set and provide confidence intervals in addition to discrete results. There are times where the model, though incorrect, did show the ability to achieve basic thresholds and get close to an answer (sometimes this might override the gold label prediction during the self-consistency step). This might provide more information beyond the direct accuracy of the model and it might provide measures in terms of model uncertainty in relation to results.

- Time and expense were two big limitations in the work that was able to be achieved. In the future, we hope to explore more rigorous comparisons of our own work with codex as well as iterate on improving the fine-tuned pipeline.

Authorship Statement

Riya: Implemented SC decoding; experimented with Jasmine on self-debugging; developed method and experiment architecture; analyzed results and wrote report

Jasmine: Implemented post-processing; experimented with Riya on self-debugging; developed method and experiment architecture; analyzed results and wrote report

Jenny: Implemented Prompt Manipulation; developed method and experiment architecture; analyzed results and wrote report

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.

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.

Shuyan Zhou Uri Alon Lengfei Liu Yiming Yang Jamie Callan Graham Neubig Luyu Gao, Aman Madaan. 2021. Pal: Program-aided language models. *arXiv preprint arXiv:2108.00648*.

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.

Yong Lin Tong Zhang Shizhe Diao, Pengcheng Wang. 2023. Active prompting with chain-of-thought for large language models. *arXiv preprint arXiv:2303.08128*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models.

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*.

A Appendix A: Prompts

A.1 Baseline Model Prompt

```
    """
Read the following passages to answer questions with Python code, store the result as
a 'ans' variable:

# Passage: James bought 93 red and 10 blue stickers, he used 31 red sticker on his
fridge and 7 blue stickers on his laptop.
# Question: How many red stickers does James have?
original_red_stickers = 93
used_red_stickers = 31
ans = original_red_stickers - used_red_stickers

# Passage: Allen went to supermarket to buy eggs, each egg costs 80 dollars, if the
discount is 29 dollars.
# Question: How much do you have to pay to buy for each egg?
original_egg_price_in_dollars = 80
discount_dollars = 29
ans = original_egg_price_in_dollars - discount_dollars

# Passage: Dianna collects both cases and books. He bought 22 cases and 5 books from
the store. Now he has 57 cases and 25 books.
# Question: How many books did danny have at first?
num_books_bought_at_store = 5
num_books_now = 25
ans = num_books_now - num_books_bought_at_store

# Passage: There were 108 chickens and 20 sheeps at the farm, some of chickens and
sheeps were sold. There are 87 chickens and 18 sheeps left now.
# Question: How many chickens were sold?
num_chicken_before = 108
num_chicken_now = 87
ans = num_chicken_before - num_chicken_now

# Passage: Katty scored 2 goals on monday, 8 goals on tuesday and 9 goals on
wednesday.
# Question: How many did Katty score on monday and wednesday?
num_goals_on_monday = 2
num_goals_on_wednesday = 9
ans = num_goals_on_monday + num_goals_on_wednesday
"""
```

A.2 Prompt for Fetching Question Type

```
    """
You are given a math word problem question. Please say whether the question involves
addition, subtraction, multiplication, or division.

Question:
Marco and his dad went strawberry picking. Together they collected strawberries that
weighed 36 pounds. On the way back Marco's dad lost 8 pounds of strawberries. Marco's
strawberries now weighed 12 pounds. How much did his dad's strawberries weigh now?
Label:
Subtraction

Question:
In a school there are 732 girls and 761 boys. 682 more girls and 8 more boys joined
the school. How many girls are there in the school now?
Label:
Addition

Question:
An industrial machine worked for 5 minutes. It can make 4 shirts a minute. How many
shirts did machine make?
Label:
Multiplication

Question:
The bananas in Philip's collection are organized into groups of size 18. If there are
a total of 180 bananas in Philip's banana collection How many groups are there?
```

```
Label:
Division
"""
```

A.3 Few-shot Prompting for Self-Debugging with Simple Feedback

```
    Read the following passages to understand how to correct python code with variable
errors:

## Start Task ##
# Passage: Tiffany was collecting cans for recycling. On monday she had 10 bags of
cans. She found 3 bags of cans on the next day and 7 bags of cans the day after that.
# Question: How many bags did she have altogether?
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_mondney + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is wrong. The error is "name 'num_bags_on_mondney' is not
defined".
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_monday + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is correct.
## End Task ##

## Start Task ##
# Passage: Tiffany was collecting cans for recycling. On monday she had 7 bags of cans.
The next day she found 12 more bags worth of cans.
# Question: How many more bags did she find on the next day than she had on monday?
num_cans_on_mondday = 7
num_cans_on_tuesday = 12
ans = num_cans_on_tuesday - num_cans_on_monday
# Feedback: The code above is wrong. The error is "name 'num_cans_on_monday' is not
defined".
num_cans_on_mondday = 7
num_cans_on_tuesday = 12
ans = num_cans_on_tuesday - num_cans_on_mondday
# Feedback: The code above is correct.
## End Task ##

## Start Task ##
# Passage: Lucy went to the grocery store. She bought 23 packs of cookie and some
packs of cake. In total she had 27 packs of grocery.
# Question: How many bags did she have altogether?
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_mondney + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is wrong. The error is "name 'num_bags_on_mondney' is not
defined".
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_monday + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is correct.
## End Task ##

"""
```

A.4 Final Prompt for Self-Debugging

```
    Read the following passages to understand how to correct python code with variable
errors:

## Start Task ##
# Passage: Tiffany was collecting cans for recycling. On monday she had 10 bags of
cans. She found 3 bags of cans on the next day and 7 bags of cans the day after that.
# Question: How many bags did she have altogether?
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
```

```
ans = num_bags_on_mondney + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is wrong. The error is "name 'num_bags_on_mondney' is not
defined".
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_monday + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is correct.
## End Task ##

## Start Task ##
# Passage: Tiffany was collecting cans for recycling. On monday she had 7 bags of cans.
The next day she found 12 more bags worth of cans.
# Question: How many more bags did she find on the next day than she had on monday?
num_cans_on_mondday = 7
num_cans_on_tuesday = 12
ans = num_cans_on_tuesday - num_cans_on_monday
# Feedback: The code above is wrong. The error is "name 'num_cans_on_monday' is not
defined".
num_cans_on_mondday = 7
num_cans_on_tuesday = 12
ans = num_cans_on_tuesday - num_cans_on_mondday
# Feedback: The code above is correct.
## End Task ##

## Start Task ##
# Passage: Lucy went to the grocery store. She bought 23 packs of cookie and some
packs of cake. In total she had 27 packs of grocery.
# Question: How many bags did she have altogether?
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_mondney + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is wrong. The error is "name 'num_bags_on_mondney' is not
defined.
num_bags_on_monday = 10
num_bags_on_tuesday = 3
num_bags_on_wednesday = 7
ans = num_bags_on_monday + num_bags_on_tuesday + num_bags_on_wednesday
# Feedback: The code above is correct.
## End Task ##

Read the following passage and correct the incorrect spelling in the variable as in
the examples above:
## Start Task
[Passage, Question and Incorrect Code]
# Feedback: The code above is wrong. The error is "name'""" + unknownVar +"""' is not
defined".
"""
```