Synthetic Instruction Tuning for Retrieval-Augmented Code Generation

Ajay Arora **Jacob Hansen** **Brian Huang**
MIT

Abstract

Despite the impressive reasoning skills of current large language models, they are still prone to errors and inaccuracies for highly knowledge-dependent tasks such as code generation. Fortunately, performance on such tasks has been shown to improve when language models are augmented with external knowledge retrieval mechanisms. Even without external memory, language models can also perform better when treated with fine-tuning or in-context learning methods that elicit reasoning in outputs. In this work, we explore the combination of these two approaches in the setting of code generation. In addition to providing autoregressive language models with a code documentation retriever, we cast these models in an in-context learning paradigm, using instances of natural language clarification for task queries as well as rationales for task question answering. The datasets used for in-context learning, which include expanded queries and rationales, are bootstrapped from GPT-3.5-turbo starting from a small seed set of human-annotated instances; as such, our method can be performed with very little human supervision. We show that various combinations of external retrieval, chain-of-thought rationales, and data augmentations can vastly improve code generation beyond the baseline performances of mid-size pretrained LLMs. Our experiments reveal the potential of co-integrating knowledge retrieval methods with the natural language reasoning of autoregressive decoder-only language models for effective usage in code generation.

1 Introduction

The scaling of large language models in recent years, particularly GPT-3 and GPT-4, has allowed for major progress on tasks requiring difficult reasoning. One crucial emergent ability of LLMs is chain-of-thought prompting (Jason Wei et al., 2022), which demonstrates high performance on tasks with or without few-shot prompting. Generation of rationales—intermediate reasoning for problem solving explicitly performed in natural language output—has proven to be valuable for LLMs in settings such as mathematics, common sense reasoning, and code generation (Nazneen Fatema Rajani et al., 2019), (Vered Shwartz et al., 2020), (Maxwell Nye et al., 2021). Another crucial ability is in-context learning (Qingxiu Dong et al., 2023). When an LLM is given exemplars of (question, answer) pairs for a task, then prompted to answer a new question in the same task, the model exhibits high performance without having been finetuned for that specific task.

Furthermore, knowledge distillation has successfully transferred reasoning capabilities from larger models to models as small as one-tenth of the original size (Rohan Taori et al., 2023), (Peng et al., 2023), and careful pretraining has also led to impressive reasoning at smaller scales (Team, 2023). Bootstrapping methods such as self-instruct (Yizhong Wang et al., 2022) or STaR (Eric Zelikman et al., 2022) provide ways to generate high-quality synthetic datasets for diverse instructions/-tasks or chains-of-thought/rationales, respectively, from the outputs of LLMs, beginning from just a few human-written exemplars. By using this synthetic data in the inputs or training loops of smaller models, distillation or transfer of knowledge and reasoning abilities into smaller models can be performed in a near-automated fashion.

Despite this plethora of abilities, large language models are often prone to errors in highly knowledge-dependent settings. One such setting, pertinent to real-world use cases today, is code generation. Human programmers naturally refer to consistently updated manuals and documentation when writing code, thereby avoiding errors. To achieve similar improvements in performance, recent research has introduced documentation retrieval into code generation pipelines for NLP models (Shuyan Zhou et al., 2023). Despite this, most specialized code generation models in literature,

such as CodeT5 (Yue Wang et al., 2021) and CodeBERT (Zhangyin Feng et al., 2020), have yet to take advantage of the raw scale and powerful reasoning capabilities of the largest language models.

In this work, we explore combinations of in-context learning, instruction bootstrapping, rationale generation and rationalization a la STaR (Eric Zelikman et al., 2022), and code documentation retrieval for the natural-language-to-code-intent task.

We give an overview of our approach through an example notated in Figure 1: given a natural language intent, we run the query through a small-scale LLM, such as a Mosaic MPT-7B model variant, fine-tuned on human annotated examples to provide an enhanced version of the query with a nuanced breakdown of the problem. We then pass this enhanced query to a black-box retriever model from (Shuyan Zhou et al., 2023) to retrieve top-k documents for this query. We then pass the query and relevant documentation to a second model (possibly the same as the first) instruction tuned on synthetic data to generate a code snippet as an answer to the original query. We perform this same pipeline on a "teacher" model, GPT-3.5-turbo, to generate a synthetic dataset usable in in-context learning on our small LLM.

We explore the effectiveness of query, retrieval, and rationale augmentation in an in-context learning setting, achieving comparable performance to other recent methods (Shuyan Zhou et al., 2023). Our approach demonstrates an impressive ability for LLMs of smaller scale to generate coherent, correct reasoning on unseen code generation problems. We also implement ablations to study the effects of each augmentation, elucidating possible shortcomings and failure modes in our code generation pipeline.

2 Related Works

Our approach is a melting pot of methods pertinent to LLM code generation, including external knowledge retrieval, in-context learning, rationale generation, and bootstrapping off of language model outputs. This section details the background for these methods and explores related approaches that have been proposed for improving code generation or general reasoning abilities in LLMs.

2.1 Reasoning in LLMs

(Jason Wei et al., 2022) introduces the chain-of-thought prompting method to enhance the reasoning abilities of large language models. By generating a rationale, a series of intermediate reasoning steps, in response to a task query, LLMs experience improved performance on complex reasoning tasks, including arithmetic, commonsense question answering, and symbolic reasoning. The intermediate steps of chain-of-thought act as a "scratchpad" for reasoning, allowing LLMs to produce higher-quality solutions. Among other settings, chain-of-thought proves particularly valuable in code generation tasks, which are consistently solvable by step-by-step reasoning from first principles. In code, even seemingly straightforward operators and functions can involve complex underlying logic that requires breaking down the problem into intermediate steps. Therefore, our approach integrates chain-of-thought into the specific setting of code generation. We aim to improve the quality of the generated code snippet by leveraging rationale generation before the model outputs its final answer.

Rationale generation is further explored in the Self-Taught Reasoner (STaR) method (Eric Zelikman et al., 2022). STaR aims to improve the performance of language models on complex reasoning tasks through iterative learning from rationale examples. First, a small number of human-written rationale examples, the "seed" examples, is used to generate synthetic rationales for a large dataset of task examples. After generation, the language model is bootstrapped—finetuned on its own synthetic rationale outputs—to refine its reasoning abilities; the model is then able to generate higher-quality reasoning on unseen examples and tasks. One crucial intervention in this bootstrap is "rationalization": to filter out lower-quality synthetic data, STaR enumerates all synthetic rationales that conclude with the wrong answer, and then regenerates each example, this time providing the correct answer as a "hint" in the original query. With rationalization, this self-teaching process steers the model in the direction of higher-correctness latent reasoning for complex problem solving. We borrow the synthetic rationale generation, complete with rationalization, from STaR to prompt our models with high-quality code generation reasoning during in-context learning. Our application of STaR differs from the original in that we focus on in-context learning without finetuning; we investigate whether synthetic rationales can "steer" our models towards more accurate code generation even without the actual bootstrapping aspect of STaR.

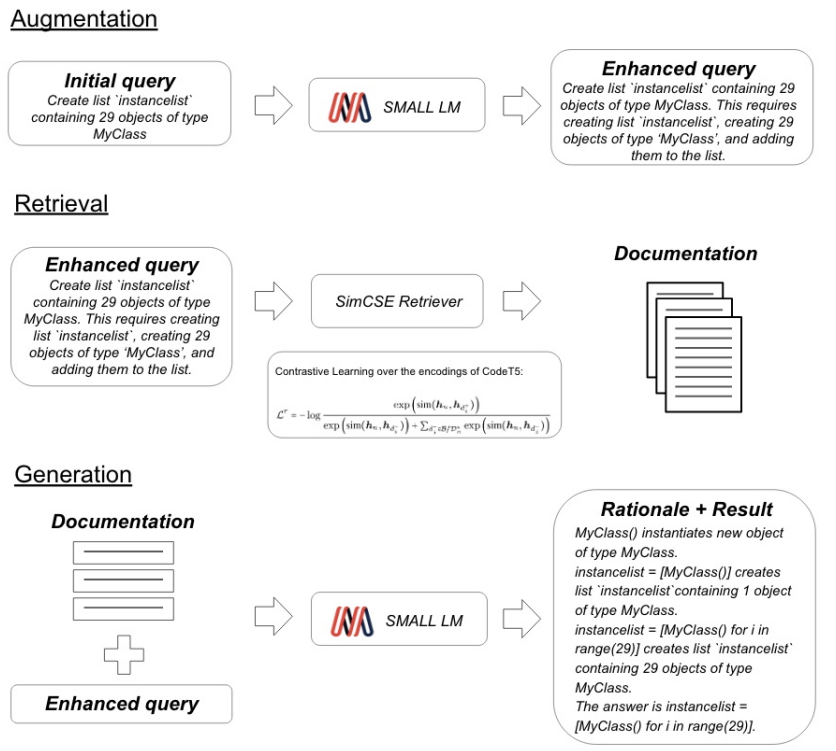

Figure 1: An overview of the instruction tuning augmented pipeline from natural language query to generated code. We indicate the three substeps of our pipeline, namely query augmentation, documentation retrieval, and code generation.

Similar bootstrapping methods have appeared alongside STaR in recent research. Self-instruct (Yizhong Wang et al., 2022) also uses a small number of human-written exemplars to "seed" a large synthetic dataset. As opposed to STaR, which examines chain-of-thought reasoning on a specific target task, self-instruct focuses instead on generating new (question, answer) pairs for unseen tasks and instructions from scratch, with the goal of bootstrapping the instruction-following capabilities of a model. In (Huang et al., 2022), an LLM is again bootstrapped on its own chain-of-thought outputs for various question-answering and reasoning tasks, but here the filtering method is self-consistency, a majority-voting mechanism on several different completions of the same prompt.

2.2 Documentation Retrieval for Code Generation

While chain-of-thought and bootstrapping methods (Jason Wei et al., 2022) (Eric Zelikman et al., 2022) lead to substantial improvements in LLM problem-solving and reasoning abilities, the specific setting of code generation poses many syntactical and knowledge issues that cannot be overcome with just improved reasoning. To aid a model's reasoning, we provide external code documentation retrieval by expanding on the work of DocPrompting (Shuyan Zhou et al., 2023). DocPrompting addresses the challenge of keeping language models up-to-date with evolving code libraries by leveraging code documentation. Their approach recognizes that human programmers frequently refer to outside textual resources such as code manuals and documentation to explore and understand available functionality. The method uses a retriever model, based on (Gao et al., 2022), to explicitly retrieve relevant documentation for a code generation query before passing this documentation downstream to a code generation model. DocPrompting thereby enhances the generation of code from natural language queries; the documentation retrieval consistently leads to improvements on code generation tasks across benchmarks such as CoNaLa, a Python-based benchmark, and tl;dr, a bash script benchmark. The integration of documentation retrieval methods in code generation offers language models the ability to generalize to unseen functions and libraries outside their training data, which would be much more difficult using just internal

knowledge and memory. Additionally, it provides a reminder to the model for nuances in syntax and updated code. Our work borrows several aspects of the DocPrompting framework; we use their adaptation of simCSE dense retrieval and benchmark on their splits of CoNaLa.

2.3 Integration of Knowledge Retrieval and Instruction Tuning

Building upon the insights from the aforementioned papers, this work combines chain-of-thought prompting (Jason Wei et al., 2022), documentation retrieval (Shuyan Zhou et al., 2023), and rationale augmentation (Eric Zelikman et al., 2022) (Yizhong Wang et al., 2022) to enhance the performance of language models in code generation. By combining these approaches, the proposed framework aims to address the limitations of existing large language models in highly knowledge-dependent tasks.

3 Methods

In this section, we expand upon the pipeline in Figure 1 and describe our usage of synthetic rationales, rationalization, and in-context learning to push further for improved code generation performance in our models.

3.1 Query augmentation bootstrapping

Consider the example query *Create list instancelist containing* 29 *objects of type MyClass*.

To inject reasoning into such queries, we consider the analogy of a teacher or teaching assistant breaking down a complex problem into its steps with a student without necessarily revealing the implementation of each step, simply with the problem statement. For the given example, the intent can be broken down into the following steps:

- creating list 'instancelist'
- creating 29 objects of type MyClass
- containing them in the list

For this example the augmented query is:

Create list 'instancelist' containing 29 objects of type MyClass.

This requires creating list 'instancelist', creating 29 objects of type MyClass, and containing them in 'instancelist'.

We adopt in-context learning for generating a synthetic dataset of enhanced queries for a query dataset Q to fine tune a smaller language model. We begin with a small seed set of human annotated mappings from intent to "enhanced intent" $P = \{(q_i, q_{ei})\}|_{i=1}^{N}$ $(N = 3)$. Similar to few-shot prompting, we concatenate P with a task description and batches of queries from Q with empty enhanced intents, which is passed to a larger pretrained model to generate enhanced intents. Once this synthetic dataset is created, we can use pairs of (q_i, q_{ei}) in the in-context learning exemplars of a code generation problem for a smaller model.

3.2 Retriever

In our work, we use a dense retriever (Shuyan Zhou et al., 2023) pretrained via a contrastive objective (Gao et al., 2022). Exact training procedures and additional information can be found in the appendix of (Shuyan Zhou et al., 2023). We note that, since our natural language queries are enhanced with additional details, it would be necessary to retrain the simCSE-based retriever for exact retrieval results in our new setting; however, for the scope of this project, it is a reasonable assumption that the original DocPrompting retriever gives a close approximation of the would-be retrained retriever.

Given an enhanced query, the dense retriever returns a list of the top-k code operators/functions for the query, accompanied by their natural language descriptions from code documentation. The functions and corresponding docs are concatenated into a "relevant documentation" part of the in-context learning prompts for our model. Note that we truncate this documentation to the first paragraph each, allowing a potentially lengthy in-context learning prompt to fit in the limited context sizes of our smaller models.

3.3 Rationale augmentation bootstrapping

Similar to query augmentation, we begin with a small-set of human annotated rationales for (enhanced query, documentation, command) tuples and use in-context learning to curate a synthetic dataset of rationales. This borrows the setup of STaR (Eric Zelikman et al., 2022). For illustration, we return to the query example introduced in Section 3.1: *Create list instancelist containing 29 objects of type MyClass* and the relevant retrieved documentation for *python range*.

We structure human-written rationales to mimic step-by-step thinking for building complex lines of code. Namely, each step is followed by a code snippet, and each additional step improves this code snippet until an answer is reached. For example:

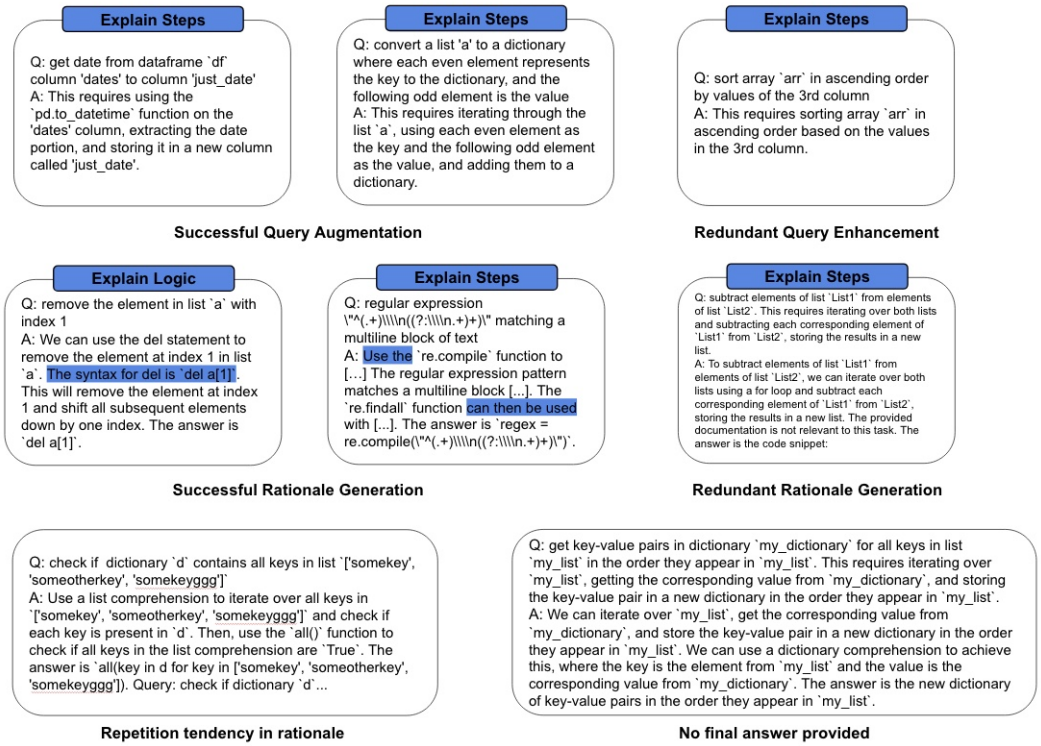

Figure 2: Examples of query augmentation and rationale generation, labelled by phenomena. Our baseline synthetic data generation approach can be brittle, with failure modes such as redundancy, parroting, or no code generation appearing occasionally in the data. As such, filtering or robustness methods like STaR-style rationalization are necessary additions to our approach.

```
1 MyClass() instantiates new object of
      type MyClass.
2 instancelist = [MyClass()] creates list
      `instancelist`containing 1 object
      of type MyClass.
3 instancelist = [MyClass() for i in
      range(29)] creates list
      `instancelist` containing 29
      objects of type MyClass.
4 The answer is instancelist = [MyClass()
      for i in range(29)].
```

After generating synthetic rationales for each query in our training set, we then borrow rationalization from STaR (Eric Zelikman et al., 2022) to enforce high-quality rationale generation. Any rationales that generate an incorrect answer on a first pass are re-run with a hint of the answer appended to the end of the enhanced query to generate accurate rationales. (While we only make one re-run attempt for each query, it's possible to further polish the quality of synthetic rationales with more rationalization passes.)

After our synthetic generation and rationalization are finished, these synthetic rationales are used for in-context learning exemplars in our query to the smaller model.

4 Experimental Setup

4.1 Data

We evaluate our framework on CoNaLa (Pengcheng Yin et al., 2018), a Python code generation benchmark collected from StackOverflow ($N = 2800$). Following (Shuyan Zhou et al., 2023), we use a train-test-split such that every example in the test set uses at least one Python function (e.g. plt.plot) that was not seen in the training data. We use the same database of Python documents ($N = 35K$, `devdocs.io`) across common Python packages.

4.2 Models

We use MosaicML MPT-7B-Chat (Team, 2023) and Stanford Alpaca-7B (Hugo Touvron et al., 2023) (Rohan Taori et al., 2023) for our small LMs and use GPT-3.5-turbo for synthetic dataset generation. All models were trained and evaluated on a 100GB A100.

We evaluate our models on the test split of DocPrompting CoNaLa. Because our models are not finetuned for code generation, their answer code snippet is encased in a larger context of natu-

ral language output. To process outputs for evaluation, we split the final sentence after "The answer is"; truncate before stop words such as "Query:" to give credit even if the model continues parroting; and strip whitespace, punctuation, and back quotes.

5 Results

We include/exclude various combinations of our four methods: in-context learning, code document retrieval, synthetic rationales, and STaR-style rationalized rationales. This constitutes a comprehensive ablation study on model performance in code generation. Our results are displayed in Table 1.

Our maximal configuration, which includes 2 in-context learning exemplars, code retrieval, and rationalized synthetic rationales, shows consistent improvement upon the baseline, where no in-context learning, retrieval, or synthetic rationales are used. This improvement is highly visible in MPT-7B-Chat; on the other hand, Stanford Alpaca-7B already shows an impressive baseline ability for code generation, so improvements from the maximal configuration as indicated by *charBLEU* and *Exact Match* are marginal, if not insignificant. Although results for the maximal configuration are ambiguous, the advantage of our approach actually becomes clear when code retrieval is ablated away. Examining just *BLEU-4* and *charBLEU*, our strongest results across all ablations occur at 2-shot in-context learning with rationalized synthetic rationales, but no retrieval. In fact, in these cases, removing retrieval improves *charBLEU* by approximately 8 points in both models, with similar increase in *BLEU-4*! Indeed, in every ablation pair with/without retrieval, the addition of retrieval negatively impacts the model's performance on all metrics.

Fortunately, for our other components of in-context learning and synthetic rationales, we experience consistent improvements when they are included. Without any other augmentations, including just two (question, answer) exemplars for MPT-7B-Chat increases *BLEU-4* by 14.33 points and *charBLEU* by 18.78 points. Even for Alpaca-7B, where the baseline performance is impressive, each of the 2-shot cases experiences a higher *BLEU-4* than the zero-shot cases.

It is worth noting that exact match is a highly brittle and biased measure compared to BLEU scores, which award partial credit. Exact match measures an absolute yes/no for correctness, which is the metric humans care more about, but this is confounded by the existence of multiple solutions for almost any code generation problem. An extension of this study might generalize exact match to *pass@k* to provide a more robust measure of correctness in code generation. (Note that exact match is equivalent to *pass@1*.)

5.1 Examining synthetic rationales

From hand-examination, our synthetic dataset is well-behaved and accurate. Namely, the vast majority of our generated rationales constitute 1-3 sentences of natural language reasoning based on the retrieval, building step-by-step on the query, culminating in a sentence that is precisely *"The answer is"* followed by a code snippet. While the formatting is stable, the outputs from GPT-3.5-turbo don't reach the exact answer very often, and only 189 out of 2117 synthetic samples have an exact match on our initial pass. STaR-style rationalization is extremely effective at compensating for the imperfect accuracy: a single pass of rationalization over the synthetic dataset regenerated over half of the examples into exact matches, resulting in 1437 exact matches in the rationalized synthetic dataset.

6 Discussion

Given the comprehensive research literature on chain-of-thought and rationale generation for improving LLM problem-solving and reasoning tasks, it's not too surprising that our cases where synthetic rationales are included in a 2-shot prompt are generally the highest-performing of the ablation grid. What may be surprising is that STaR-style rationalizing the synthetic dataset actually does improve upon the original synthetic dataset, even for our limited in-context learning application. This improvement is not as significant for Alpaca-7B, but MPT-7B-Chat experiences a 3.48-point increase in *charBLEU* upon synthetic rationale rationalization. Our results inadvertently provide a slight piece of evidence against the viewpoint in (Sang Michael Xie et al., 2022), who claim that in-context learning helps a language model locate the latent concepts for an input task, rather than learn to answer the task from the exemplar answers. For our code generation setting, it does improve performance to have higher accuracy in the final answers of our in-context learning exemplars.

The apparent importance of correctness and relevance of in-context learning exemplars also hints

Mosaic MPT-7B-Chat

Ablation	BLEU-4	charBLEU	Exact Match
Zero-shot No augmentation	11.58	12.3	1.11
Zero-shot Retrieval	5.99	7.69	0.0
2-shot No augmentation	25.91	31.08	**4.26**
2-shot Rationale	25.77	29.67	2.22
2-shot Retrieval	14.24	15.5	1.48
2-shot Retrieval and rationale	22.58	26.65	2.22
2-shot STaR rationalized	**26.45**	**33.15**	3.7
2-shot Retrieval STaR rationalized	21.67	25.28	1.11

Stanford Alpaca-7B

Ablation	BLEU-4	charBLEU	Exact Match
Zero-shot No augmentation	18.73	25.24	1.85
Zero-shot Retrieval	13.18	15.6	1.3
2-shot No augmentation	21.17	23.98	2.04
2-shot Rationale	27.91	**33.59**	2.59
2-shot Retrieval	20.98	22.9	2.04
2-shot Retrieval and rationale	19.61	23.54	2.22
2-shot STaR rationalized	**28.37**	**33.62**	**3.33**
2-shot Retrieval STaR rationalized	21.05	25.66	2.04

Table 1: Ablation results from evaluating Mosaic MPT-7B-Chat and Stanford Alpaca-7B on CoNaLa with various combinations of our augmentations. We use the pretrained retriever from (Shuyan Zhou et al., 2023) with top-3 retrieved docs. BLEU-4 measures BLEU scores with tokenization and maximum 4-gram order, while charBLEU uses language-agnostic character-level tokenization. Best results for each column are bolded.

towards possible explanations for the poor performance of retrieval. The top-3 documents returned by our simCSE retriever can be error-prone and highly noisy for the setting of code generation. It is not uncommon that some of the top-3 documents are irrelevant to the answer code snippet, and/or that some functions in the answer code snippet are not returned by the retriever. Even when all retrieved functions are pertinent to the query, the accompanying documentation can contain spurious details unrelated to the answer code snippet. As such, the retrieval in its current iteration may often mislead or otherwise confound the model ability to reason smoothly to the correct code snippet.

Besides correctness and relevance issues, the retrieved documentation is also very lengthy, typically taking up the majority of tokens spanning an exemplar. Both of our models have a limited 2048-token max context, and inclusion of retrieval can easily take the input above 1000 tokens even for 2-shot cases, so the sheer length of retrieval may what is be degrading model performance. In fact, we performed preliminary evaluations with 3 or 4 in-context learning exemplars and found that the extra exemplars led to degenerate model outputs much more often than using just 2 exemplars. Moreover, the prompt with 4 exemplars occasionally exceeded 2048 tokens, making the prompt unusable in our models.

6.1 Limitations

Our evaluated models are on the smaller end of LLMs at only 7B parameters, and they are both pretrained on general, diverse text. The limitations of this regime of models for code generation, which combines real-world domain knowledge with complex step-by-step reasoning, are made more clear by our experiments. The MPT-7B family includes code tokens as 10% of all pretraining tokens, and sources this data from The Stack, BigCode's code corpus (Team, 2023); on the other hand, the amount of code pretraining in Alpaca-7B is unclear.

We note that *Exact Match* results are particularly low across both MPT-7B-Chat and Alpaca-7B, while GPT-3.5-turbo is able to generate a significantly higher percentage ($8.93\% = 189/2117$) of exact matches for the synthetic dataset even before rationalization. Although GPT-3.5-turbo likely dwarfs our 7B models in parameter size, this observation points out a pitfall in our use of chain-of-thought. It is well-known from (Jason Wei et al., 2022) that below certain model sizes, chain-of-thought reasoning fails to improve performance and may actually hurt performance. Indeed, our 7B models may be at the borderline of size where chain-of-thought may not clearly improve performance. Future work should explore larger parameter models with CoT and larger context size models where the volume of retrieval content and the nature of retrieval-augmented reasoning may not inherently hurt the computation.

7 Conclusion and Future Directions

Our study explored the combination of external code retrieval, in-context learning, and synthetic data augmentations such as bootstrapped rationales to improve code generation performance. We conducted experiments with our prompting and code parsing setup on two small instruction-finetuned language models. We elucidated the potential for in-context learning, bootstrapped synthetic rationales, and rationalization to synergize and facilitate impressive performance on code generation, even for small-scale LMs not specialized for code generation. At the same time, our experiments exposed the challenging nature of applying external knowledge retrieval for prompting of LLMs.

All of our experiments were done without our own fine-tuning, so it remains to be seen whether finetuning our models for code generation may yield improvements on top of our current approach.

Future work could expand our human-annotated exemplars for our synthetic datasets and tune the prompts used during training, as this may enhance the model's reasoning capabilities. To evaluate the generated code more comprehensively and fairly credit the wide range of correct code solutions, a benchmark based on dynamic Python environments with test cases could provide more reliable metrics and enable a closer examination of the code's functionality and correctness. At the very least, our approach may be evaluated on more diverse distributions of code beyond the Python-based CoNaLa.

The rationale augmentation pipeline proposed here can be applied to many external knowledge-based problems beyond code-generation. This includes but is not limited to language translation, dialogue agents, technical question answering tasks. Generally, we hope for this work to serve as a foundation for injecting reasoning into retrieval-augmented tasks.

References

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. STaR: Bootstrapping Reasoning With Reasoning. *NeurIPS*.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2022. Simcse: Simple contrastive learning of sentence embeddings.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. *arXiv preprint arXiv:2112.00114*.

Nazneen Fatema Rajani, Bryan McCann, Caiming Xiong, and Richard Socher. 2019. Explain Yourself! Leveraging Language Models for Commonsense Reasoning. *ACL*.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. *IEEE/ACM MSR*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A Survey on In-context Learning. *arXiv preprint arXiv:2301.00234*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. *GitHub repository*.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2022. An Explanation of In-context Learning as Implicit Bayesian Inference. *ICLR*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating Code By Retrieving The Docs. *International Conference on Learning Representations (ICLR)*.

MosaicML NLP Team. 2023. Introducing mpt-7b: A new standard for open-source, ly usable llms. Accessed: 2023-03-28.

Vered Shwartz, Peter West, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2020. Unsupervised Commonsense Question Answering with Self-Talk. *EMNLP*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Model with Self Generated Instructions. *arXiv preprint arXiv:2212.10560*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *EMNLP*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *EMNLP*.