Abstract

Program Synthesis from Natural Language using Language Models

Ansong Ni

2024

Programming is a ubiquitous problem-solving tool and a generic method of instructing machines. However, mastering programming skills can take months or even years of practice. Thus, making programming more accessible has been a key problem in computer science since its inception. More specifically, generating programs directly from human inputs (e.g., natural language instructions) has been a longstanding challenge in artificial intelligence, due to the ambiguity of natural language and the precision required in programming.

Recent advances in large language models (LLMs) have shown great potential in this area. These LLMs, typically neural networks with billions of parameters, are trained with trillions of tokens of text and code, equipping them with the ability to understand natural language and generate code. With their capability for program synthesis from natural language, such LLMs can empower real-world applications such as virtual personal assistants, AI pair-programming, robotics control, and provide natural language interfaces for data queries and visualization.

While LLMs have greatly pushed the frontier of program synthesis from natural language inputs and achieved state-of-the-art performance on various code generation benchmarks, their coding capabilities still lag significantly behind those of human programmers. To further improve the capabilities of LLMs in synthesizing programs from natural language inputs, several challenges need to be addressed: 1) LLMs are data- and compute-hungry, making it expensive to train or finetune such models; 2) Although LLMs are adept at generating plausible code, they typically lack the ability to model and reason about program execution; 3) There is a lack of comprehensive evaluation of the language-to-code generation abilities of LLMs.

Addressing the difficulties in obtaining large amounts of training data, we first present a self-training framework for program synthesis using LLMs, where we use the model to

sample additional programs to augment the ground truth program for learning. This involves an iterative process of program sampling, execution-based filtering, and learning from the correct ones stored in the cache. Moreover, we propose partial correctness and demonstrate that we can further improve the learning efficiency and model performance by learning not only from fully-correct self-sampled programs but also from partially-correct ones.

To equip the LLMs with the ability to reason about program execution, an essential skill for complex coding tasks such as debugging, we train the LLMs to understand execution traces and generate natural language chain-of-thought reasoning rationales. We first design an LLM-friendly trace representation and then reuse the self-training framework, prompting the model to first generate natural language rationales and then the code outputs. While we only filter by the correctness of the code outputs, this typically also filters out low-quality rationales. With iterative training on high-quality rationales and the correct code outputs, we show that LLMs can improve their ability to reason about program execution. Such rationales mimic the reasoning process of human programmers and can also be used to directly communicate with users to improve the interpretability of LLMs for coding tasks such as program repair.

To address the issue of the huge computational cost of finetuning large models to understand program execution, we demonstrate that it is possible to train a smaller model as a verifier to assess the program candidates sampled from the LLMs for each natural language input. These verifiers learn to verify the correctness of the generated programs along with their execution results, and rerank the program samples based on the generation probability from the LLMs and the verification probability from the smaller verifier models.

Lastly, to obtain a comprehensive understanding of the language-to-code generation capabilities of LLMs, this dissertation also presents an evaluation on 7 benchmarks from three key domains: semantic parsing, math reasoning, and Python programming. We evaluate a total of 56 models of various sizes, training data mixtures, and training methods, with the aim to study the effect of these factors on model performance across different tasks. Targeting the data contamination issue of code generation benchmarks, we also present a pipeline to identify such contaminated examples with both surface- and semantic-level searches, and quantify the level of contamination for two commonly used datasets.

Program Synthesis from Natural Language using Language Models

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Ansong Ni

Dissertation Director: Arman Cohan

December, 2024

In loving memory of Drago, a great mentor, a good friend, and a kind man.

Contents

List of Figures

List of Tables

Acknowledgements

Whenever I read the acknowledgements in other people's theses, I would wonder what it would be like and what I would write when my turn came – and now, here we are.

First and foremost, I would like to thank my late advisor Dragomir Radev and my current advisor Arman Cohan. I still remember my first exchange with Drago. It was February of 2020, and I had just gotten off an international flight. I opened my email, only to find three more PhD rejection letters. With Yale being one of the few schools hadn't rejected me yet, I emailed Drago to try to make a case for myself and immediately got the reply telling me not to worry as he has already selected me for admission. It turned out in the end that Drago wass the only one to admit me out of the 15 schools and dozens of professors I applied to. So when I say I owe my whole career to Drago and I couldn't have imagined where I would be right now if it weren't for him, I'm simply stating a fact. I also remember my last exchange with him, it was March of 2023, I got an email from Drago telling me that he was sorry that he couldn't make it to an external talk I was giving and hoped that it went well. I definitely could not remember all my interactions with Drago throughout the first two and a half years of my PhD, but his endless passion for research, and his constant optimism towards life had made a great impact on me, for which I will carry with throughout my career and my life. I'm also lucky to have Arman Cohan as my advisor, who has been nothing but supportive of me and my research. During the difficult times, it was Arman who stepped up and took responsibilities for me and the lab, for which I will be forever grateful. I have learned a great deal from Arman for the past year and a half, especially his attention to details in research and his approaches in creating a great atmosphere for this new lab. This dissertation would not have been possible without the support and care from Arman.

I would also like to thank my other committee members, Graham Neubig, Bob Frank and Ruzica Piskac. Graham is another person that has great impact in my career. During my master's days at CMU, Graham opened the door for me to the world of NLP research. I still don't know what Graham saw in me to take me, someone who knew nothing about NLP back then, as a student to do research with. But it was during my project with Graham when I decided to keep pursuing this research direction and start doing a PhD. I couldn't have asked for better committee members than Bob and Ruzica, who are experts in the fields of computational linguistic and programming languages, respectively. You always provide invaluable feedback from different perspectives, and encourage me to get out of my comfort zone and I would always learn a lot every time I do so.

My PhD research is also largely shaped by the four research internships I have done over the years. I would like to thank Pradeep Dasigi and Matt Gardner for the mentorship during my time at AI2, and helping me publish my first paper in NLP conferences. I am also grateful for my time spent at MSR, (virtually) working alongside Alex Polozov, Jeevana Priya Inala, Chenglong Wang, Chris Meek and Jianfeng Gao. It was during this internship that I decided to commit to using large language models for program synthesis, and I learned a lot about perseverance and resilience in approaching challenging research problems. I also greatly appreciate Victoria Lin, Sida Wang, Scott Yih, Ves Stoyanov and Srini Iyer for the super fun project we did during my internship at Meta AI. It was my first onsite internship and I have a wonderful time in Menlo Park, it also helped me realize that industrial research labs are where I want to go after my PhD. Last but certainly not the least, I would like to thank Pengcheng Yin, Charles Sutton, Miltos Allamanis, and Kensen Shi for the time during my internship at Google DeepMind. The L4C team is really an all-star team in the field of AI for code, as I was constantly surrounded and inspired by these extremely talented and also incredibly kind people. Pengcheng is also a great collaborator throughout the years, starting with my first NLP project at CMU. Pengcheng always takes less credit than he deserves, I remember when Pengcheng "edited" my paper the first time at CMU, he basically re-wrote a couple of sections and then he emailed me and Graham that "Ansong did a great job in writing and I only need to do a few minor edits". I am really blessed to have a chance to learn from all the great researchers I had the honor of working with

throughout these internships.

I would also like to thank my mentors and collaborators in other research projects of mine, including Ahmed Awadallah, Asli Celikyilmaz, Budhaditya Deb, and Chenguang Zhu for the collaboration on the summarization project with MSR; Ye Liu, Semih Yavuz, Caiming Xiong, Shafiq Joty, and Yingbo Zhou for the L2CEval project in collaboration with Salesforce Research. Besides Graham, I would also like to thank some more of my previous mentors and managers, who helped me reaching this stage, these include my undergrad research advisor, Ming Li, my mentor for my internship project at CMU, Claire Le Goues and Ruben Martins, and my manager for my very first internship, Shi Han.

I am also very fortunately to be surrounded by the best lab mates and colleagues. During my first year, I had the chance to learn from the seniors in our labs as Rui Zhang, Tao Yu, Alex Fabbri, and Irene Li. Although it was during Covid and I didn't get to see all of you face-to-face until a couple of months into my PhD, you helped me quickly adapt to the PhD life and filled me with hope for the years to come. I also learned a great deal from my peers, Linyong Nan, Yixin Liu, Borui Wang, Simeng Han, Yilun Zhao and Kaili Liu. All the hotpots and bbq chickens we had, I am sure that those have become a part of me that I will carry with (literally). I would like to especially thank Linyong and Yixin, for listening to my nonsense throughout the years, and for all the fun nights we had appreciating the magic of hydroxyethane. I also had the chance to work with some of the most brilliant undergrads at Yale, including Zhangir Azerbayev, Martin Riddell, Hailey Schoelkopf, Ziming Mao, Mutethia Mutuma, Troy Feng, and Stephen Yin. I would also like to thank the other student collaborators I have worked with over the years, Yinlin Deng, Yusen Zhang, Chen Wu, Rui Shen, Daniel Ramos, and Aidan Yang.

I have also been blessed with lots of friends outside of the lab, and I have not forgotten about you! It was not easy for me to leave my family and friends in China, and move all the way to the US just by myself. It was especially hard for my PhD time since the first year and a half of it were entirely spent during the pandemic. Luckily, I soon found myself surrounded by a group of interesting, caring and sometimes annoying friends. I would like to thank (alphabetically so none of you complains) Kate Candon for the chitchats over the lunches; Nick Georgiou for all the sports we played and watched; Siddharth Mitra for always

making it impossible to make group plans; Luis Ramirez for your signature jokes; Marco Pirazzini for burning all the the midnight oil with me in that corner of ours; and Morgan Vanderwall for the great hikes. I also want to thank the friends in the YINS gang: Jane Lee, Iason Ofeidis, Ioannis Panitsas, Grigoris Velegkas, Xifan Yu and Felix Zhou; as well as friends in the CS department: John Lazarsfeld, Anay Mehrotra, and Argyris Oikonomou. I am also grateful for my teammates and friends in the SEAS IM soccer and basketball teams, for still taking me despite how bad I am. I had a lot of fun playing with you and thanks for putting some physical exercise in me. Together we won three championships, which really adds a lot of bragging power in the sports bracket for me. Okay, I really hope I haven't missed anyone, but right now it's 4:37AM in the morning so I'm sure you all can understand if I did.

Finally, I would like to thank my parents and Ziyue Zoey Yang for their unconditional love and support, I owe all my success to them. My mom made it her full-time job taking care of me a few years after I was born, and was my private tutor helping me with homework till middle school. My dad travels all the time due to his work, so we cherish every moment we spent together as a family. I met Zoey in high school, and she has been my greatest counselor, firmest supporter, and best friend ever since. We attended colleges in two cities that are hundreds of miles apart; during Covid, we didn't see each other for more than 9 months; and there was a year when we lived in opposite sides of the earth. However, her support and care never wavered, always empowering me to face challenges, knowing that someone had my back. Her dedication and perseverance in her own work also constantly inspires me to push my limits and achieve more. She is the smartest, most considerate, and caring person I have ever met, for which I feel incredibly lucky every single day.

Chapter 1

Introduction

1.1 Program Synthesis from Natural Language

Program synthesis – the task of automatically generating computer programs that fulfill the users' requirements – has been a long-standing challenge in artificial intelligence and computer science (Backus et al., 1957; Manna and Waldinger, 1971; Summers, 1977; Waldinger and Lee, 1969). Such task is especially important as of modern day since we are surrounded by computing devices and yet the way to instruct them is still mostly restricted to writing computer programs, which is a skill that most people may not possess.

While program synthesis in broader terms considers also sorts of human inputs that construct the specification, such as input-output pairs (Devlin et al., 2017; Gulwani, 2016), in this thesis, we put focus on **program synthesis from natural language** (Desai et al., 2016; Lin et al., 2017a; Raza et al., 2015), as natural language (*e.g.,* English, Spanish, Mandarin) is how we communicate with each other for the vast majority of humanity. With the technology to generate programs from natural language, people will be able to directly communicate with machines and instruct them to facilitate our work and daily life, without the need to learn or use programming. To illustrate this, here we show some examples of tasks that can be solved by generating programs in Figure. 1.1. For example, when we would like to know the answer to the question of *"Who avg the most pts this season?"*, we can use a program synthesis model to automatically generate a SQL query, "`SELECT name FROM players ORDER BY avg_pts LIMIT 1`". Then such SQL query will be executed

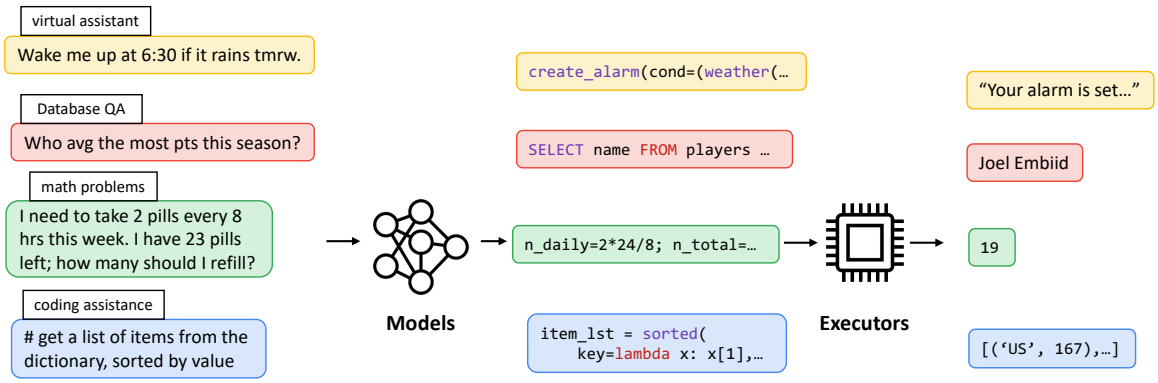


Figure 1.1: Many real-world tasks can be approached by generating programs from natural language inputs, and those programs are later executed to obtain the desired results.

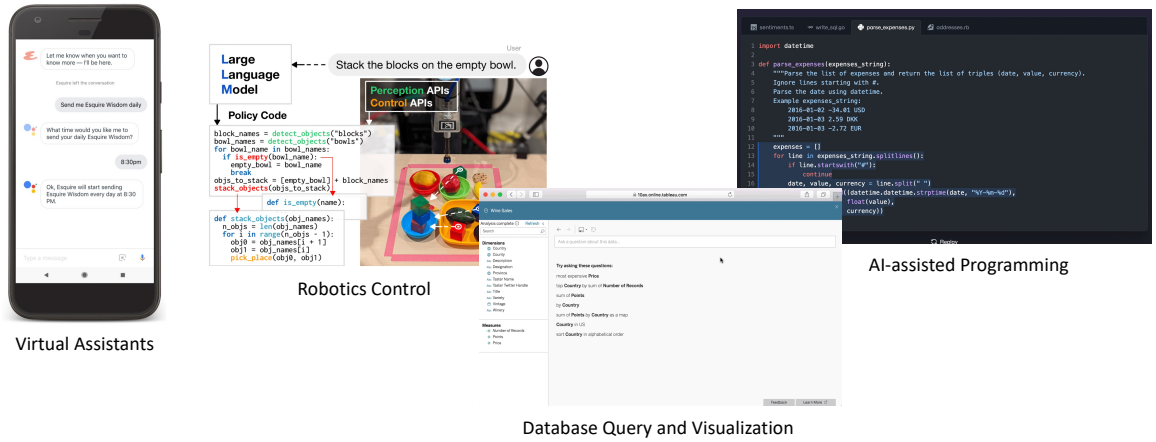


Figure 1.2: Program synthesis models power many real-world applications.

on a player statistics database to obtain the answer for the user. In a different example, a program synthesis model can also be used as a coding assistant to generate general-purpose programs (*e.g.*, Python) as “`item_lst = sorted(key=lambda x: x[1], item_dict`” from a comment written in natural language, “`# get a list of items from the dictionary, sorted by value`”. Given the expressiveness of this task formulation, such program synthesis models can also be the backbone of many more real-world applications, such as robotics control shown in Figure. 1.2.

While the technology of program synthesis from natural language could be very impactful, carrying out such tasks also faces several key challenges. **(1) Ambiguity of natural language instructions:** Compared with programs which are written in formal languages, natural language instructions are usually ambiguous and under-specified. Take the first example in Figure. 1.1 for instance, in the utterance “*Wake me up at 6:30 if it rain tmrw*”,

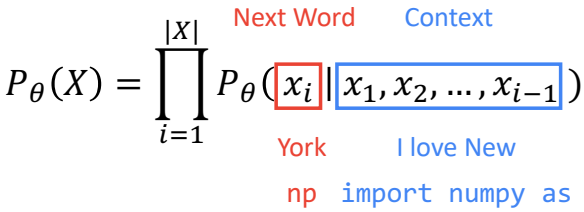


Figure 1.3: Formulation of a typical (*i.e.,* auto-regressive) language model.

it is not specified whether it is "6:30 AM" or "6:30PM", which creates an ambiguity that needs to be resolved with commonsense reasoning (*i.e.,* people usually wake up in the morning and not in the afternoon). **(2) Programming requires absolute precision.** Since programs are meant to be interpreted and executed by the computers, there is rarely any room for error. This makes the program synthesis tasks much harder to make or even measure progress than natural language generation tasks. For example, randomly corrupting a character in a natural language sentence will most likely not affect its semantic meaning when read by other people, but it is very likely to cause a compilation error which renders the whole program unusable. **(3) Large search space.** Since programs follow strict grammatical rules, thus it is possible to perform exhaustive search for short completions with simple grammar (Solar-Lezama et al., 2005, 2006). However, the search space still grows exponentially with the length of the programs, thus it is intractable to search over longer programs with complex grammar rules.

1.2 Challenges for Large Language Models on Code

As illustrated in Figure. 1.3, a language model learns to predict the next work given the context. Recent advances in building large language models (LLMs) show that such models are able to perform multiple tasks using the same model (Radford et al., 2019) and they can be *prompted* to perform new tasks with in-context learning (Brown et al., 2020). One of the greatest benefit brought by the pre-trained language models is a tractable way to model the probability of sequences of arbitrary length by autogressively predicting a probability distribution over the vocabulary. While it is possible to use grammar rules to classify the all possible next tokens as valid or invalid (*i.e.,* 0 or 1), LLMs gives a continuous probability on the next tokens, making it possible to optimize the search and easier to perform sampling.

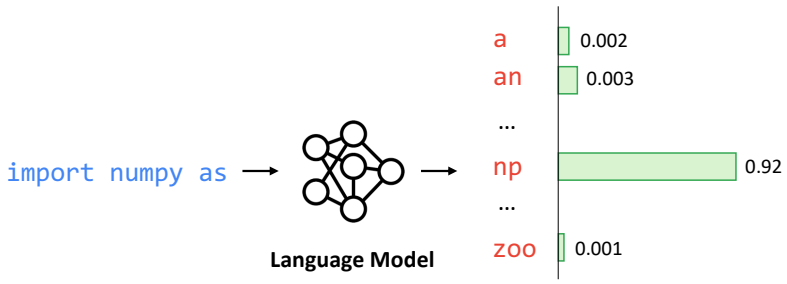


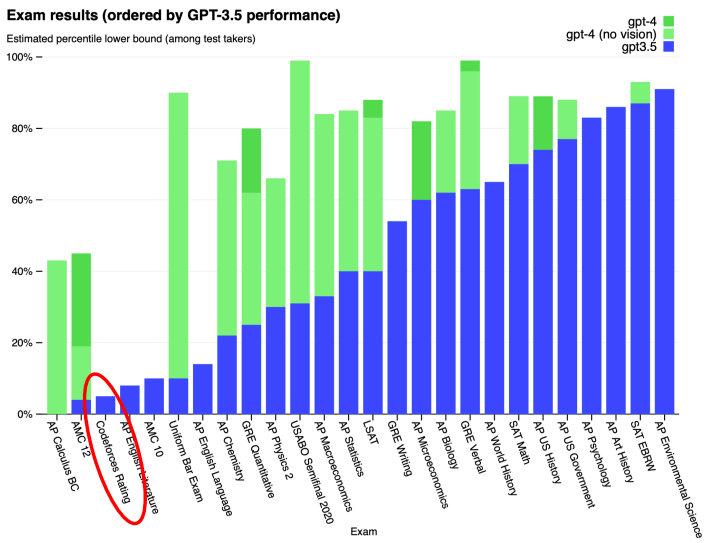
Figure 1.4: Pre-trained language models greatly reduce the search space for programs.

Moreover, as shown in Figure. 1.4, a well-trained language model would set the probabilities of most tokens to zero, especially those that are not grammatically correct. Such a pre-trained language model not only has a "built-in" syntax model, it also learns the coding styling and conventions through pre-training, such as predicting "`np`" after "`import numpy as`".

Another built-in advantage of such LLMs is that they are typically trained on a mixture of data that is from different sources and modalities, including natural language and code, which naturally empowers them to generate programs from natural language inputs (Anil et al., 2023; OpenAI, 2023; Radford et al., 2019; Touvron et al., 2023a). Such coding capabilities are further enhances if they are trained on more tokens on code or code-related data (*e.g.,* documentation) (Fried et al., 2022; Li et al., 2023; Nijkamp et al., 2022; Roziere et al., 2023). Thus the task of generating programs from natural language can be simply modeled as a sequence to sequence generation problem using the LLMs without further training. Moreover, LLMs are often equipped with adequate (commonsense) reasoning abilities to disambiguate (without context) and generate the correct programs.

While the emergence of large language models greatly advances the field of program synthesis by providing a strong base model, it also poses new challenges for further advancing their abilities to generate more complex programs. As shown in Figure. 1.5, while GPT-4 scores at high percentiles for tests as SAT and GRE, its coding capability is below 5th percentile compared with human programmers by the codeforces[1] ratings. Here we discuss the challenges incurred for using LLMs to approach the problem of program synthesis from natural language inputs.

1. `https://codeforces.com`



Simulated exams	GPT-4 estimated percentile
Uniform Bar Exam (MBE+MEE+MPT)[1]	298/400 ~90th
LSAT	163 ~88th
SAT Evidence-Based Reading & Writing	710/800 ~93rd
SAT Math	700/800 ~89th
Graduate Record Examination (GRE) Quantitative	163/170 ~80th
Graduate Record Examination (GRE) Verbal	169/170 ~99th
Graduate Record Examination (GRE) Writing	4/6 ~54th
USABO Semifinal Exam 2020	87/150 99th–100th
USNCO Local Section Exam 2022	36/60
Medical Knowledge Self-Assessment Program	75%
Codeforces Rating	392 below 5th

Figure 1.5: The performance of GPT-4 on various tasks, the coding scores are circled in red. Source: `https://openai.com/index/gpt-4-research/`

LLMs are data- and compute-hungry. The training of large language models requires a lot of data to generalize well at test time, and larger models also typically need more training data to unlock their potentials (Hoffmann et al., 2022; Kaplan et al., 2020). While researchers found that more code data generally leads to better coding capabilities for LLMs (Guo et al., 2024; Ni et al., 2023b), high quality code data is also not as abundant as text since it is typically produced by professionals (*i.e.,* programmers). Moreover, a large quantity of code data is privately held or licensed with restrictive licenses (*e.g.,* GPL, GNU) thus not contributing to training of open-source models. At the same time, the training of these large language models also incurs huge cost in compute, which is can be approximated by the formula of $FLOPS \approx 6 \times \text{model\_size(B)} \times \text{training\_tokens(B)}$ (Kaplan et al., 2020). As an example, the BLOOM 176B model (Le Scao et al., 2023) is trained on 384 NVIDIA A100 80G GPUs for 3.5 months, consuming more than 1 million compute hours. And finetuning such models would not be possible without at least 2,112GiB of GPU memory by estimation.[2] This greatly limits the post-training efforts (*e.g.,* fine-tuning, reinforcement learning) that could further improve the coding capabilities of LLMs, thus a new learning or data creating paradigm is needed for code generation. Moreover, the cost also greatly limits

2. Estimated by FP32 full-parameter tuning using formulation from `https://blog.eleuther.ai/transformer-math/`

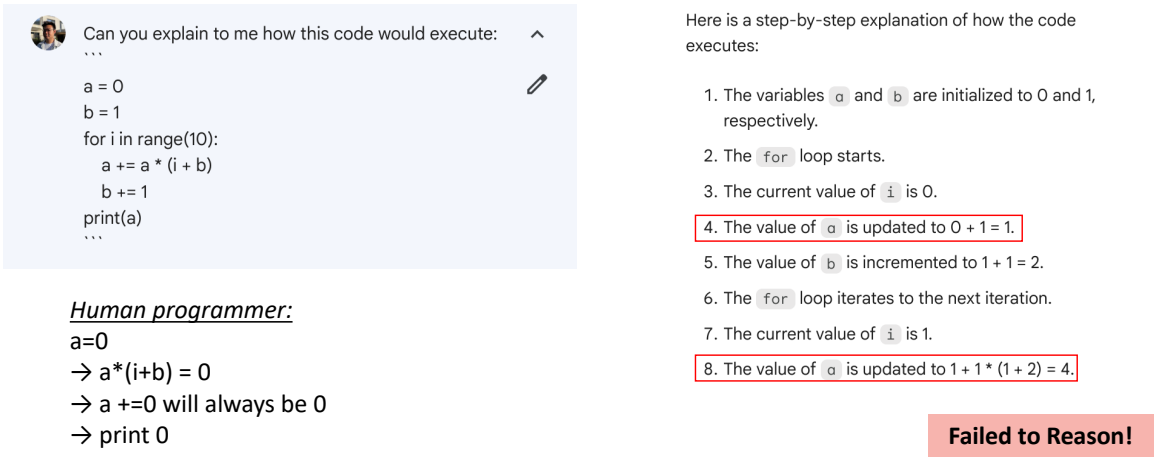


Figure 1.6: Example conversation with Google Bard, compared with how human programmers would reason about program execution. *Accessed 2023-08.*

the accessibility for training, fine-tuning or even simply testing such LLMs, especially for small businesses and academic labs.

Modeling program execution. Most of the LLMs obtain their coding capabilities by including publicly available codebases from websites as GitHub[3] in their pre-training data mixture (Li et al., 2023; Roziere et al., 2023; Touvron et al., 2023a). Such process makes them very good at writing plausible code, which is great for applications such as code completion, but recent research also found that LLMs do not really understand how the code works, *i.e.,* they are unable to reason about program execution at runtime like human programmers (Gu et al., 2024a,b; Ni et al., 2024). And example of this is shown as Figure. 1.6, which such a simple program is easy for human programmers to reason about, it turns out to be quite difficult for LLM-based conversation agents to perform the same task. While the LLMs are only trained on the *surface textual form* of the code, the *semantics* of the a program lies in its execution, which the models have very limited exposure to at training time. The inability to model and reason about execution will prevent LLMs from obtaining human-level programming skills and greatly affect the future of applications such as coding assistants.

Evaluating LLMs for language-to-code generation. While language-to-code generation is a general tool for solving problems as math reasoning, text-to-SQL parsing, etc, as

3. `https://github.com`

shown in Figure. 1.1, such problems are usually separately studied, leading to fragmented understanding of how such LLMs can understand different natural language instructions and generate task-specific programs (*e.g.,* SQL, Python, etc). Moreover, while most of the language models obtain their coding capabilities through training on a mixture of data that contains code, we have a limited understanding of how factors as model size, percentage of code data, and prompting methods affect the performance of LLMs on natural language to code generation tasks. At the same time, since the LLMs are typically trained on a mixture of data scraped from the Internet from different sources, test data contamination is an issue that is usually overlooked when evaluating the capabilities of LLMs (Deng et al., 2023; Jiang et al., 2024b). And this data contamination problem is arguably more severe for code generation as the solutions to the popular programming testing websites can often be found on websites like GitHub (Jain et al., 2024; Lozhkov et al., 2024). Such issues make it hard to evaluate the progress of LLMs on the problem of language-to-code generation and formulate a clear understanding of the limitations for current methods.

1.3 Contributions

In this dissertation, we aim to address some the aforementioned challenges to accelerate the progress in using large language models to approach the problem of program synthesis from natural language.

Learning with self-sampled programs. Targeting on the data-hungry challenge of LLMs for program synthesis, we propose to perform self-sampling and learning from self-sampled programs in Chapter 3 and self-sampled natural language rationales in Chapter 4. This is based on the observation that program correctness can be easily evaluated given test cases or expected outputs, thus we can build a "sample→filter→train" loop to iteratively improve models' performance on code-related tasks. More specifically, in Chapter 3 we propose the self-training framework for math programming tasks and . Moreover, we define partial correctness and propose ways to find partially correct program prefixes for programs that do not yield the correct final results. Experiments on two math reasoning datasets show

the effectiveness of our method compared to learning from a single reference solution with MLE, where we improve PASS@100 from 35.5% to 44.5% for GSM8K, and 27.6% to 36.2% PASS@80 for MathQA. More broadly, we show that the models can benefit from learning from alternative programs that the models sampled themselves, setting the stage for the research in self-training for LLMs for code. Moreover, we smooths the definition for program correctness by proposing partial correctness, which makes it easier to perform self-training for programs.

Reasoning about code execution. To equip LLMs with the ability to reason about program execution like human programmers, we introduce NEXT in Chapter 4, a method to teach LLMs to inspect the execution traces of programs (variable states of executed lines) and reason about their run-time behavior through chain-of-thought (CoT) rationales. Similar to the methods in Chapter 3, NEXT uses self-training to bootstrap a synthetic training set of execution-aware rationales that lead to correct task solutions (*e.g.,* fixed programs) without laborious manual annotation. Experiments on program repair tasks based on MBPP and HumanEval demonstrate that NEXT improves the fix rate of a PaLM 2 model, by 26.1% and 14.3% absolute, respectively, with significantly improved rationale quality as verified by automated metrics and human raters. We also show that NEXT can also generalize to scenarios where program traces are absent at test-time. This work is the first to propose a method that can reliably improve the ability of LLMs to reason about program execution. As LLMs are used as agents to operate in the real-world (Wang et al., 2024a; Xi et al., 2023), generating and execution code is an essential way for such agents to take actions. It is imperial for such LLM-based agents to reason about the program behavior before taking actions, and this work is a solid step towards that goal.

Learning from execution feedback. While fine-tuning the LLMs on the self-sampled examples is an effective way of improving LLMs' coding ability and teaching them to model and reason about code execution, directly fine-tuning such LLMs also incurs a huge cost as mentioned in Figure. 1.2. In Chapter 5, we propose LEVER, a simple approach to improve language-to-code generation by learning from execution feedback without directly fine-tuning

the LLMs themselves. More specifically, we use the LLMs as black boxes to produce program samples and train smaller models as verifiers to determine whether a program sampled from the LLMs is correct or not based on the natural language input, the program itself and its execution results. The sampled programs are reranked by combining the verification score from the smaller verifier and the generation probability from the LLMs, and marginalizing over programs with the same execution results. On four datasets across the domains of table QA, math QA and basic Python programming, LEVER consistently improves over the base CodeLMs (4.6% to 10.9% with `code-davinci-002`) and achieves new state-of-the-art results on all of them. LEVER shows that it is possible to improve large language models by training smaller models as verifiers to rerank the outputs. More importantly, we show that we can inject execution semantics into these smaller verifier models to improve the whole language model system for program synthesis.

Systematic evaluation of language-to-code generation for LLMs. To obtain a comprehensive understanding of the ability of LLMs to generate code from natural language inputs, we present `L2CEval` in Chapter 6. `L2CEval` is a systematic evaluation of the language-to-code generation capabilities of LLMs on 7 tasks across the domain spectrum of semantic parsing, math reasoning and Python programming, analyzing the factors that potentially affect their performance, such as model size, pre-training data, instruction tuning, and different prompting methods. In addition, we assess confidence calibration, and conduct human evaluations to identify typical failures across different tasks and models. Later in this chapter, we also discuss a study of data contamination of popular code generation benchmarks, and precisely quantify their overlap with pre-training corpus through both surface-level and semantic-level matching. With experiments, we show that there are substantial overlap between popular code generation benchmarks and open training corpus, and models perform significantly better on the subset of the benchmarks where similar solutions are seen during training. Chapter 6 offers a comprehensive understanding of the capabilities and limitations of LLMs in language-to-code generation, and is valuable for building future LLMs and evaluation benchmarks for code generation.

1.4 Outline

The following chapters in this dissertation are directly adapted from the following publications:

- **Chapter 3: "Learning Math Reasoning from Self-Sampled Correct and Partially-Correct Solutions".** In The Eleventh International Conference on Learning Representations (ICLR), 2023.
- **Chapter 4: "Teaching Large Language Models to Reason about Code Execution".** In The Forty-first International Conference on Machine Learning (ICML), 2024.
- **Chapter 5: "LEVER: Learning to Verify Language-to-Code Generation with Execution".** In The Fortieth International Conference on Machine Learning (ICML), 2023.
- **Chapter 6: "L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models".** In The Transactions of the Association for Computational Linguistics (TACL), 2024; **"Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models".** In The 62nd Annual Meeting of the Association for Computational Linguistics (ACL).

The resources (*e.g.,* code, data, slides) for this dissertation can be found at `https://github.com/niansong1996/PhD-Thesis`

Chapter 2

Background

In this section, we discuss the background information for this dissertation from two aspects, the task of program synthesis and the development of language models. While we do not comprehensively cover the history of these topics, we aim to provide a good understanding of the field to better understand the main contributions in the following chapters.

2.1 Program Synthesis

In this section, we discuss the background of program synthesis, by first giving its task formulation then introduce a series of related tasks to show how people approach the program synthesis problem, and lastly discuss how program synthesis systems are typically evaluated.

2.1.1 Task Formulation and Evaluation

In this dissertation, while we broadly define program synthesis as the generating computer programs that satisfy certain specifications, we focus on natural language descriptions as the main format of such specifications.

Task Formulation. As shown in Figure. 2.1, given a natural language input x (*e.g.,* "*How many students in the class are between 20 and 30 years old?*"), a program synthesis model generates a candidate program $\hat{y}$, *e.g.,* "`SELECT COUNT(name) FROM students where age > 20 AND age < 30`". Such program may also be executed given the execution context ϕ (*e.g.,* a

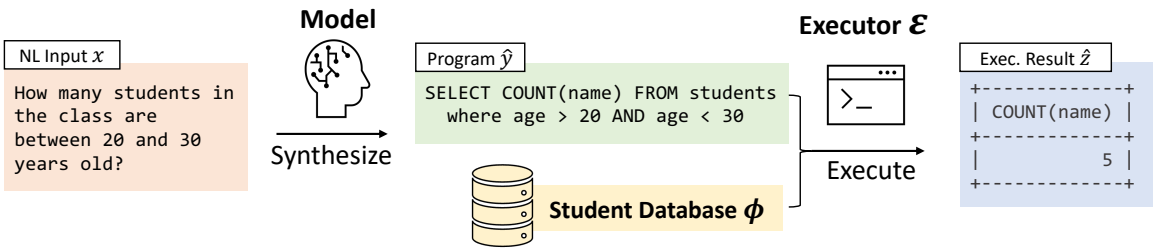


Figure 2.1: The formulation for program synthesis from natural language inputs, using text-to-SQL generation as an example.

database) and an executor $\mathcal{E}(\cdot)$ to produce an execution result $\hat{z}$, *e.g.,* "`COUNT(name) = 5`".

Surface-form evaluation and its limitations. Given the model generated program candidate $\hat{y}$, we can evaluate its quality by comparing it with the gold standard program y^*. The *exact match* metric simply compares $\hat{y}$ and y^* on the string-level after some normalization or canonicalization. While such evaluation is straightforward and creates almost no false-positive, exact match could lead to significant false-negative issues, especially for grammatically rich programming languages as the same specification can be satisfied by semantically equivalent programs with different surface forms (*e.g., "multiply var with itself"* can be implemented with "`var = var * var`" or "`var = var ** 2`" in Python). And it is also possible to write algorithmically different programs that achieves the same goals, *e.g.,* bubble sort and merge sort. Thus exact match is only commonly used for programs that are less flexible, such as λ-calculus (Hemphill et al., 1990; Yin and Neubig, 2018), CCGs(Artzi and Zettlemoyer, 2013; Zettlemoyer and Collins, 2005), or sometimes SQL queries (Yin and Neubig, 2017; Yu et al., 2018b). Another category of surface-form evaluation for programs are based on n-grams, using metrics as BLEU (Papineni et al., 2002) or ROUGE (Lin, 2004). While such metrics works well for natural language generation, and can be used to measure progress when models are relatively weak (Wang et al., 2022c; Yin et al., 2018b), researchers found that such n-gram based metrics correlates poorly with the correctness of the programs (Ren et al., 2020). Model-based evaluation of the surface-form is also developed for code generation, yielding higher correlations with functional correctness (Zhou et al., 2023). Another issue with surface-form evaluation is that expert annotations are typically required to write the reference programs y^*.

Execution-based Evaluation. While surface-form-based evaluation focuses on matching the generated programs $\hat{y}$ to the references y^*, execution-based evaluation skips the generated programs themselves and purely rely on the matching of the execution results $\hat{z}$ and expected execution results z^*. This is often referred to as the *execution accuracy* (Yin and Neubig, 2018; Yu et al., 2018b). While this metric directly measures the correctness of the program candidate $\hat{y}$ generated by the model, since it is based on a single program candidate per natural language input x, it typically leads to a large variance when estimating the model performance, especially when we have a small test set or the performance is low. In solving this problem, the PASS@k metric (Kulal et al., 2019) has gain significant popularity recently: instead of relying on a single program output per task (*i.e.,* natural language input), we obtain k program candidates from the synthesizer for each problem x, and if *any* of those k programs are correct, we set PASS@$k = 1$ for this problem. Then we estimate the performance by taking the average PASS@k on the test set. The reason why PASS@k only cares about the *existence* of correct programs given k samples is because for code generation applications, it is generally reasonable to assume access to test cases which can easily prune out most incorrect program candidates; but if there are not any correct programs in the candidates, there is nothing post-hoc that we can do. To further reduce the variance of the PASS@k metric, Chen et al. (2021a) proposed to use n samples where $n > k$, and proposed an unbiased estimator to calculate the PASS@k performance, shown as Equation. 2.1

$$\text{PASS@}k := \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{2.1}$$

While the calculation of the binomial coefficient can be numerically unstable, Chen et al. (2021a) also shared an numpy implementation that makes the calculation of Equation. 2.1 more stable.

It is also worth noting that while the current evaluation for program synthesis almost exclusively focus on functional correctness, researchers are also starting to pay attention to other aspects of the synthesized programs, such as efficiency, latency and maintainability (Shypula et al., 2023; Singhal et al., 2024).

2.1.2 Related Tasks

Program synthesis from structured inputs. The pursuit of automated program synthesis is tied with the development of *compilers* in the early days, which can be traced back to 1950s (Backus et al., 1957), with the FORTRAN compiler translating from higher-level FORTRAN language into lower-level assembly. There are also some early attempts for automatically construct programs from specifications written as input and output predicates (Manna and Waldinger, 1971; Waldinger and Lee, 1969). A resurgence of interest for program synthesis techniques appeared with the concept of *programming by example (PBE)*, which aims to automatically induct the programs from a set of "$<$ *input*, *output* $>$" pairs. An example of PBE is FlashFill (Gulwani, 2011), which synthesizes string processing programs that map from multiple string inputs to an output string. The synthesis process of PBE typically involves searching within the grammar of a domain-specific language (DSL). With the advancement of deep learning, there are also attempts in *neural program synthesis* which uses neural networks to directly output the programs or accelerate the search (Bunel et al., 2018; Chen et al., 2018a; Devlin et al., 2017; Nye et al., 2020b).

Semantic parsing. Semantic parsing aims to transduce natural language utterances into formal representations. Different from syntactic parsing, semantic parsing extends beyond grammatical rules to understand the meaning of the sentences. While early semantic parsing systems use rule-based pattern matching (Johnson, 1984; Woods, 1973b), they are quite brittle given the flexibility and ambiguity of natural language sentences. Later, statistic-based learning approaches are proposed to develop more robust systems, as Zelle and Mooney (1996b) developed the CHILL algorithm to convert natural language sentences into database queries. Works as Zettlemoyer and Collins (2005) and Artzi and Zettlemoyer (2013) proposed algorithms to parse the natural language sentences into combinatory categorical grammar (CCG). Similar to the field of program synthesis, neural-based approaches were also later developed for semantic parsing (Dong and Lapata, 2018; Jia and Liang, 2016; Krishnamurthy et al., 2017; Yin and Neubig, 2017), followed by a surge of interests in text-to-SQL parsing due to its evident potential for building natural language interfaces for databases (Yu et al., 2018a,b; Zhong et al., 2017b). Different learning paradigms also emerged for semantic

parsing, such as weakly-supervised learning (*aka* learning from denotations) (Dasigi et al., 2019a; Goldman et al., 2018,?) and active learning (Ni et al., 2020; Sen and Yilmaz, 2020).

2.2 Language Models for Code

As shown in Equation. 2.2, a language model parameterized by θ learns to predict the next token x_i given the previous context $\{x_1, x_2, ..., x_{i-1}\}$, and it is also often referred to as an auto-regressive language model.

$$P_\theta(X) = \prod_{i=1}^{|X|} P_\theta(x_i|x_1, x_2, ..., x_{i-1}) \tag{2.2}$$

In broader terms, it can also predict the token in the middle given the context before and after this token, which is useful for tasks as representation learning (Mikolov et al., 2013) and infilling (Bavarian et al., 2022; Fried et al., 2022). While this formulation is typically used to model natural language, people soon found that we can train these language models to model source code in the exact same way (Brown et al., 2020; Chen et al., 2021a).

2.2.1 Data Collection

Here we introduce the typical process for data collection for training LLMs on code, including the collection, filtering and cleaning of the data.

Data sources. The data sources for training LLMs for code are usually from GitHub or similar open-source platforms (Li et al., 2023; Roziere et al., 2023; Touvron et al., 2023a). Recent works also found it to be beneficial to learn from code-related artifacts, such as software documentation, issues and pull requests, as well as some intermediate representations such as those created by LLVM (Lozhkov et al., 2024).

Data filtering. After the raw data is collected, which are usually organized by different code repositories, certain repository-level quality filtering is usually in place to make sure that the models are trained on high-quality code data. For example, SantaCoder (Allal et al., 2023) uses several heuristics for filtering out low-quality repositories, such as the

number of stars and the comment-to-code ratio. Moreover, we also need to pay attention to the license for each repository to make sure that the collected data is only repositories with permissive licenses (*e.g.,* MIT, Apache 2.0) (Fried et al., 2022; Kocetkov et al., 2022).

De-duplication and de-contamination. The main reason for data de-duplication is because researchers found that repeated training data can significantly hurt the model performance (Groeneveld et al., 2024; Hernandez et al., 2022; Tirumala et al., 2024). For code data, people typically use hashing for exact match (Chen et al., 2021a; Fried et al., 2022; Nijkamp et al., 2022), or employ near-deduplication methods such as edit distance (Anil et al., 2023) or MinHash (Xu et al., 2022). Decontamination, on the other hand, aims to remove files that contain solutions to benchmarks that are later used for evaluating the trained model. The aforementioned methods for de-duplication can also be used for de-contamination, and in addition to those, semantic matching is also used for the purposes of measuring data contamination for source code (Riddell et al., 2024).

2.2.2 Model Architecture

Similar to general purpose language models (*i.e.,* language models for natural language), language models for code can also be categorized by the encoder-only, encoder-decoder, and decoder-only architectures.

Encoder-only models. Encoder-only models, *e.g.,* BERT (Devlin et al., 2019b) and RoBERTa (Liu et al., 2019), are usually used to learn sentence representations and can often be finetuned to perform classification tasks. Similarly for encoder-only models for code, they are typically used to perform code representation learning to facilitate tasks such as code retrieval, such as CodeBERT (Feng et al., 2020). One benefit for modeling code is that code is naturally multi-modal (*i.e.,* surface-form, abstract syntax tree (AST), control flow graph, dependency graph, execution traces) and it is usually automatic to obtain other modalities from source code. Many encoder-only code LMs take advantage of this to jointly encode different modalities of code to learn better code representation model, such as Code-MVP (Wang et al., 2022a) and GraphCodeBert (Guo et al., 2020). As mentioned in

§ 2.1.2, text-to-SQL parsing is a popular type of semantic parsing tasks, as it can be used to solve tasks as table/database question answering, and various encode-only pretrained models also emerge for table understanding, such as TAPEX (Liu et al., 2021), TaBERT (Yin et al., 2020), and GraPPa (Yu et al., 2020).

Encoder-decoder models. The encoder-decoder models, *e.g.,* BART (Lewis et al., 2019), T5 (Raffel et al., 2020), are typically trained with a mixture of classification and generation task. For encoder-decoder models for code, people also use a mixture of code-specific tasks to train such models, such as identifier tagging and identifier prediction in CodeT5 (Wang et al., 2021), and code generation and summarization in PyMT5 (Clement et al., 2020). Compared with encoder-only models, encoder-decoder models can also do generation tasks which retaining the ability to perform classification using mostly only the encoders. However, since it requires a clear separation of input and output, it is more suitable for tasks as code translation or code summarization, and less suitable for tasks as code completion. But also due to the separation of the encoder and the decoder, it is also more memory-efficient than the decoder-only models.

Decoder-only models. The decoder-only models, which are popularized by the GPT series (Brown et al., 2020; Radford et al., 2018, 2019), are language models that only use the decoder part of the transformer (Vaswani et al., 2017) and model the entire sequence autoregressively. Early models as CodeGPT (Lu et al., 2021a) show that pretraining on code data can significantly improve its performance on code generation tasks over baselines such as GPT-2 (Radford et al., 2019). The decoder-only architecture is also the more popular architecture for modeling code, with models such as StarCoder (Li et al., 2023), CodeLLaMA (Roziere et al., 2023), CodeGen (Nijkamp et al., 2022), and InCoder (Fried et al., 2022), due to its versatility and native support for tasks as code completion. In order to boost the coding capabilities while maintaining most of the text ones, some of the code-specific language models are trained on top of the general-purpose LMs. For example, the Code LLaMA model (Roziere et al., 2023) is trained on top of LLaMA 2 (Touvron et al., 2023b) with additional 500B tokens on code. Multi-stage training is also typically to

produce code LMs that are better for certain programming languages (Nijkamp et al., 2022; Roziere et al., 2023).

Chapter 3

Self-Training for Program Synthesis with Language Models

In this chapter, we present the self-training method for language models on math reasoning tasks to improve their performance during the supervised finetuning (SFT) stage. More specifically, we propose to let the models not only learn from the reference solutions that are provided in the training data, but also attempt to sample alternative solutions that are different from the reference ones but also yields the correct results. To alleviate the issues of binary learning signal from program correctness, we also define partial correctness, and show that the models can benefit from not only the fully-correct alternative solutions, but also ones that are *partially-correct*, which yields correct intermediate program states but not the final result. Experiments are conducted on two math reasoning tasks, for which we approach by generating straight-line Python programs to solve. And results show that our proposed self-training method greatly improves PASS@k performance for language models, especially when k is large. Our code is available at `https://github.com/microsoft/TraceCodegen`. [1]

1. The content of this chapter is directly apapted from *"Learning Math Reasoning from Self-Sampled Correct and Partially-Correct Solutions"* by Ni *et. al* , 2023.

3.1 Introduction

Recent progress on pretrained language models (LMs) shows that finetuned LMs are able to achieve human-level performance on various natural language processing tasks (Brown et al., 2020; Devlin et al., 2019a; Raffel et al., 2020). However, such LMs still lack the ability to perform multi-step math reasoning even for problems that are intended for grade-school students (Cobbe et al., 2021). Current methods for solving math problems typically rely on generating solutions (a sequence of computation steps) and executing them to obtain the final answer (Austin et al., 2021b; Chen et al., 2021b; Chowdhery et al., 2022; Cobbe et al., 2021), as directly generating the final answer would require computational abilities that even the largest LMs do not possess (Brown et al., 2020; Chowdhery et al., 2022).

When finetuning such LMs on math reasoning, existing methods often rely on the MLE objective that aims to maximize the log-likelihood of the reference solution for each natural language input. However, in addition to the reference solution, there are often multiple correct solutions for each question, resembling alternative reasoning paths to the final answer. However, those alternative solutions are unseen during training, and this results in model overfitting: the model becomes overly confident in its predictions because it sees the same solution over multiple epochs of training (Austin et al., 2021b; Bunel et al., 2018; Cobbe et al., 2021). This leads to poor generalization on unseen inputs and is reflected by the low PASS@k performance, where the model is unable to predict the right answer even when allowed multiple attempts per question.

To mitigate this issue, we propose learning from self-sampled solutions. Concretely, during training time, the model samples alternative solutions, and keeps track of all solutions that are semantically correct with respect to the gold execution result, and learns from all of these correct solutions as opposed to only from the reference. To further improve the effectiveness of learning from self-sampled solutions, we allow the model to learn from partially-correct solutions, whose intermediate states are consistent with intermediate states of known correct solutions. This new technique allows the model to maximally utilize the self-sampling and more efficiently explore the solution space. We also study various common loss functions for learning from multiple targets for a single natural language input, including

augmented-MLE, Maximize Marginal Likelihood (MML) and β-smoothed MML (Guu et al., 2017b) and find that their different gradient equations greatly affect the learning capabilities of the model.

We perform experiments on two math reasoning tasks, namely MathQA-Python (Austin et al., 2021b) and Grade-School-Math (GSM) (Cobbe et al., 2021), and finetune GPT-Neo models (Black et al., 2021) to generate Python program as solutions from the problem description in natural language. Results show that learning from self-sampled solutions can improve the PASS@100 from 35.5% to 44.5% for GSM, and 27.6% to 36.2% for PASS@80 on a filtered version of MathQA-Python.[2] Moreover, we find that learning from partially-correct solutions generally improves performance over learning from just fully-correct solutions (*e.g.,* +3.0% PASS@100 for GSM8K) as it guides the sampling process, discovering more alternative solutions for learning. Such performance boosts from our proposed methods are also consistent for different model sizes for the LMs. Ablation on different loss functions shows that MLE-Aug loss is the most effective in learning from multiple targets and yields the most improvements over MLE loss.

3.2 Overview

Problem formulation. We consider the task of generating solutions from math problem descriptions in natural language (NL). Given an NL input $x \in \mathcal{X}$ and the executor $\mathcal{E} : \mathcal{Y} \to \mathcal{Z}$, the goal is to generate a solution $y \in \mathcal{Y}$ that executes to the expected answer $z^* \in \mathcal{Z}$, *i.e.,* $\mathcal{E}(y) = z^*$.

Standard approach and its limitation. The standard approach is to assume that we have a dataset of paired NL input x and reference solution y^*. Most datasets typically only provide one reference solution for a particular NL input. Then, a parameterized model P_θ is learned with the *Maximum Likelihood Estimation* (MLE) objective from the NL-Solution pair (x, y^*) as:

$$\mathcal{L}_{\text{MLE}}(x, y^*, P_\theta) = -\log P_\theta(y^*|x) \tag{3.1}$$

2. We choose different k for evaluating PASS@k to be consistent with previous work.

The builtin assumption of using Equation. 3.1 for learning is that only the reference solution y^* is correct. However, this assumption is clearly untrue for the math reasoning problem as typically multiple reasoning paths can achieve the correct final result. With only one reference solution as target for learning, Equation. 3.1 would encourage the model to put all probability mass on y^*, which could easily lead to *overfitting* (Austin et al., 2021b; Bunel et al., 2018; Cobbe et al., 2021).

Overview of our approach. While manually collecting additional reference solutions for each specification is a laborious process (Austin et al., 2021b; Cobbe et al., 2021; Schuster et al., 2021), in our work, we explore an alternate approach: where the model self-samples additional correct (or partially-correct) solutions and learns from them during training. Figure. 3.1 shows an example: for the question x, our model was able to self-sample an alternative solution $\hat{y}$ that is different from the reference solution y^* provided in the dataset. Looking at the intermediate states shown on the right, we can see that both these solutions execute to produce the sample desired output, *i.e.,* $\hat{z} = z^*$, as noted with solid red boxes. Taking this one step further, our approach can also identify partially-correct solutions from its samples. For example, on the bottom left, we show a sampled solution $\hat{y}'$ that is incorrect only because of an error in its last two steps. But we identify a prefix $\hat{y}'_{\leq 5}$ of it as partially-correct because the intermediate state $\hat{s}'_5$ for this prefix matches the intermediate state s^*_5 of a known correct solution y^* (noted as dashed red boxes) and yet syntactically different from y^*. Based on these observations and intuitions, we introduce our approach in the following sections.

3.3 Learning from Self-Sampled Solutions

We now formally present our approach. There are three main steps: 1) *sampling* 2) *filtering* and 3) *learning* as shown in Algorithm. 1. Here we mainly introduce the self-sampling framework using only fully-correct solutions and the extensions with partially-correct solutions will be introduced in § 3.3.3.

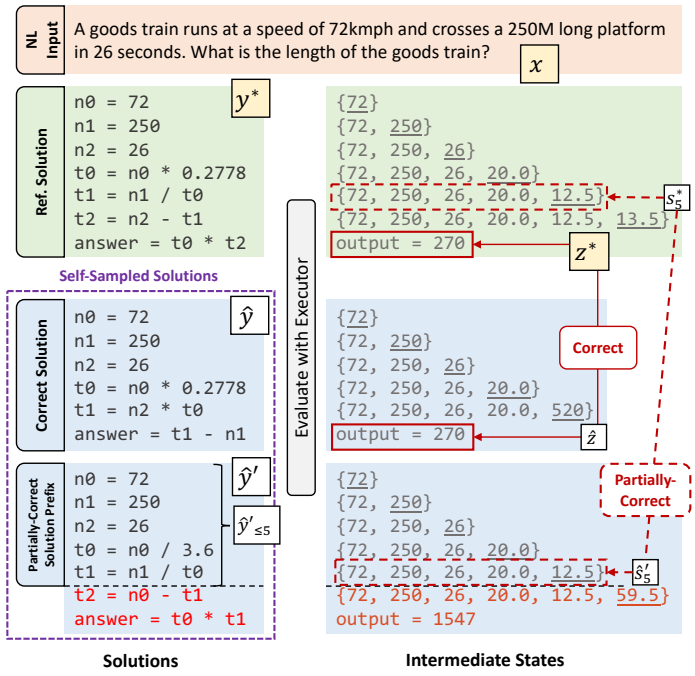


Figure 3.1: Examples of self-sampled correct and partially-correct solutions from MathQA (more in § 3.4.7). The steps and intermediate states marked in red are *incorrect*.

Algorithm 1 Training Update

Input:
Parameterized model $P_\theta(y|x)$;
Executor $\mathcal{E} : \mathcal{Y} \to \mathcal{Z}$;
A training example (x, y^*, z^*);
Buffer $\mathcal{B}$ for this input x

1: **if** $|\mathcal{B}| = 0$ **then**
2: $\quad \mathcal{B} \leftarrow \mathcal{B} + \{y^*\}$ /* initialize buffer */
3: **end if**
4: $\hat{Y} \leftarrow SampleSolutions(x, P_\theta, \mathcal{B})$
5: **for** $\hat{y}$ **in** $\hat{Y}$ **do**
6: $\quad \hat{z} \leftarrow \mathcal{E}(\hat{y})$ /* execute solution */
7: $\quad$ **if** $isCorrect(\hat{z}, z^*)$ **then**
8: $\quad\quad$ **if not** $isDuplicate(\hat{y}, \mathcal{B})$ **then**
9: $\quad\quad\quad \mathcal{B} \leftarrow \mathcal{B} + \hat{y}$ /* save to buffer */
10: $\quad\quad$ **end if**
11: $\quad$ **end if**
12: **end for**
13: $\theta \xleftarrow{\text{update}} \nabla_\theta \mathcal{L}(x, \mathcal{B}, P_\theta)$

3.3.1 Online Sampling and Filtering

For each specification x, we maintain a buffer $\mathcal{B}$ to save the different solutions that are correct, *i.e.,* evaluate to the correct result. Note that the buffers are persistent and cumulative across training epochs. To add more solutions in $\mathcal{B}$, we perform online sampling and filtering as follows.

Online sampling (line 4 in Algorithm. 1): With the NL question x from each example (x, y^*, z^*) as input, the model samples a set candidate solutions $\hat{Y} = \{\hat{y}_i\}_{i=1}^n \sim P_\theta(\hat{y}|x)$;

Filtering incorrect solutions(line 7 in Algorithm. 1): As not all sampled solutions in $\hat{Y}$ are correct (thus not suitable for learning), we filter out all incorrect solutions in $\hat{Y}$, *i.e.,* $\hat{Y}^* = \{\hat{y}|\hat{y} \in \hat{Y}; \mathcal{E}(\hat{y}) = z^*\}$;

Filtering duplicate solutions (line 8 in Algorithm. 1): Because the model can sample solutions that are correct but are "trivial variants" of other already saved solutions (*e.g.,* the solution differs from another solution only in white spaces, comments or trivial steps like "`x = x * 1.0`"), we further filter the buffer to remove them. This is essential as all saved

Name	Loss Functions $\mathcal{L}(x, \mathcal{B}, P_\theta)$	Gradients $\nabla_\theta(x, \mathcal{B}, P_\theta)$
MLE	$-\log P_\theta(y^*\|x)$	$-\nabla_\theta \log P_\theta(y^*\|x)$
MLE-Aug	$-\sum_{\hat{y}\in\mathcal{B}} \log P_\theta(\hat{y}\|x)$	$-\sum_{\hat{y}\in\mathcal{B}} \nabla_\theta \log P_\theta(\hat{y}\|x)$
MML	$-\log \sum_{\hat{y}\in\mathcal{B}} P_\theta(\hat{y}\|x)$	$-\sum_{\hat{y}\in\mathcal{B}} \frac{P_\theta(\hat{y}\|x)}{\sum_{\tilde{y}\in\mathcal{B}} P_\theta(\tilde{y}\|x)} \nabla_\theta \log P_\theta(\hat{y}\|x)$
β-MML	$-\frac{1}{\beta} \log \sum_y P_\theta(\hat{y}\|x)^\beta$	$-\sum_{\hat{y}\in\mathcal{B}} \frac{P_\theta(\hat{y}\|x)^\beta}{\sum_{\tilde{y}\in\mathcal{B}} P_\theta(\tilde{y}\|x)^\beta} \nabla_\theta \log P_\theta(\hat{y}\|x)$

Table 3.1: Comparison of loss functions and their gradients over multiple reference $\mathcal{B}$. Note that they all degenerates to MLE when only the gold reference solution is used as target, *i.e.,* $\mathcal{B} = \{y^*\}$.

solutions will be directly used for learning and such undesired behavior from the model will be encouraged without the filtering process.[3] Concretely, we first perform filtering based on the linearized abstract syntax trees (ASTs) to eliminate the differences in white space, etc; then we set a constraint on maximum number of lines using the number of lines in y^* as the reference to prevent saving solutions with trivial steps.

3.3.2 Learning from Multiple Targets

With self-sampling, each natural language question is paired with multiple solutions as targets for learning. Here we discuss some common loss functions for the multi-target learning problem, with a focus on how each target contributes to the gradient. The loss functions and their gradients are shown in Table. 3.1.

Augmented MLE (MLE-Aug): This objective augments MLE with multiple targets simply by summing the loss from multiple solutions in $\mathcal{B}$, which is equivalent as minimizing the KL-divergence from $P_\theta(y|x)$ to $Q(y|x) = \frac{1}{|\mathcal{B}|} \cdot \mathbb{1}_\mathcal{B}(y)$, where $\mathbb{1}(\cdot)$ is a set indicator function. It encourages the model to put equal weights on all targets by ensuring that all targets equally contribute to the gradient.

Maximum Marginal Likelihood (MML): MML attempts to approximate $P_\theta(z^*|x)$ by marginalizing over the correct solutions in $\mathcal{B}$. However, for each target $\hat{y} \in \mathcal{B}$, the gradient of it is in proportion to the likelihood $P_\theta(\hat{y}|x)$ given by the model, which results in a positive

3. Our preliminary experiments also show that the performance greatly degenerates when such trivial variants are left in the buffer for learning.

feedback loop during gradient updates. It encourages the model to still put a majority of the probability on one of the solutions in $\mathcal{B}$ as noted in (Guu et al., 2017b).

β-smoothed MML (β-MML): Proposed in (Guu et al., 2017b), the β-MML objective is an extension of MML with a hyperparameter $\beta \in (0, 1]$ to adjust weights of the gradient from each target. It an interpolation between MML and MLE-Aug objectives, more specifically, it recovers MML when $\beta = 1$ and its gradient is equivalent to that of MLE-Aug when $\beta \to 0$.

Empirically, we find that these distinctions between those loss functions greatly affects the model performance (Figure. 3.3), especially when partially-correct solutions are included for learning.

3.3.3 Learning from Partially-Correct Solutions

Besides learning from self-sampled *fully-correct solutions* (FCSs), we can also let the model learn from *partially-correct solutions* (PCSs). Our motivation is that the model often encounter solutions that are close to being correct as they only make mistakes in the last few steps (*e.g.,* Figure. 3.1), and these partially-correct solutions provide additional learning opportunities. Learning from PCSs could also address the issue that the sampler may have a low chance of encountering fully-correct solutions for complex tasks due to the sparse solution space.

Identifying Partially-Correct Solutions

When the model samples a solution that does not produce the desired answer, we want to identify if a prefix of this solution is partially correct, *i.e.,* it performs some of the necessary computation steps needed for the correct solution, so that the model can additionally learn from these potentially unseen prefixes in the next iteration. A challenge here is figuring out when a prefix is partially correct. Ideally, we want to say a prefix $y_{\leq i}$ is partially correct if there exists a suffix $y_{>i}$ such that their concatenation $(y_{\leq i}||y_{>i})$ is a correct solution. There are two caveats here: (1) if there is no length restriction on the suffix, it is always possible to find a suffix that complements any prefix (*e.g.,* a full gold solution is one such suffix); and (2) it is computationally very expensive to search for all suffixes (even with a length

restriction) to check if a prefix can be completed to a correct solution.

To overcome these challenges, we leverage the gold reference solutions and any self-sampled fully-correct or even partially-correct solutions to help identify new partially-correct prefixes. The idea is to identify a prefix as partially correct if it produces a set of intermediate values (upon execution) that exactly matches the set of intermediate values produced by a prefix of a known correct or partially-correct solution. For such a prefix, we know that there exists a reasonable complement suffix based on the suffix of the known solutions. Note that, this definition of partial correctness is conservative compared to the ideal definition above, but it makes the computation significantly tractable.

Below, we formally define this notion of partial solutions that leverages existing known fully and partially correct solutions.

Intermediate state. Given a solution $y = (u_1, ..., u_t)$ where u_i is the i-th reasoning step, we define the intermediate state s_i as the set of all variables values in the scope after executing the first i steps $y_{\leq i} = (u_1, ..., u_i)$, which we call a *prefix* of this solution. It is easy to see that the prefixes $y_{\leq i}$ and intermediate states s_i of a solution construct a bijective function, which is also illustrated in Figure. 3.1. Note that the state representation is name-agnostic since variable names do not typically contributes to the semantics of the solutions.

State-based equivalence and partial correctness. Given the definition of the intermediate state, we say the prefixes of two solutions, $y_{\leq i}$ and $y'_{\leq j}$, are *semantically equivalent* if and only if $s_i = s'_j$, *i.e.,* those two solutions produces the exact same set of variable values. And then we define *partial correctness* as follows: a solution prefix $y_{\leq i}$ is partially-correct if and only if it is semantically equivalent to the prefix of another known partially-correct solution $y^*_{\leq j}$. As we keep all known partially-correct solutions in the buffer $\mathcal{B}$, formally:

$$\mathit{PartiallyCorrect}(y_{\leq i}) \iff \exists y^* \in \mathcal{B}.\ \ \exists j \leq |y^*| \ s.t.\ s^*_j = s_i$$

Algorithm 2 *SampleSolutions*$(x, P_\theta, \mathcal{B})$ with partially-correct solutions

Input: Model $P_\theta(y|x)$; the NL input x and a set of partially-correct solutions $\mathcal{B}$
Output: Solution samples $\hat{Y}$.

1: Select $\hat{y}_{\leq i} \in \mathcal{B} \setminus \{\hat{y} | \mathcal{E}(\hat{y}) = z^*\}$ uniformly at random /* sample PCS prefix for completion */
2: Sample a set of completions $Y_p \sim P_\theta(\hat{y}_{>i} | \hat{y}_{\leq i}, x)$
3: $\hat{Y} \leftarrow \{[\hat{y}_{\leq i} || \hat{y}_{>i}]\}_{\hat{y}_{>i} \in Y_p}$ /* concatenate completions with the solution prefix */
4: **return** $\hat{Y}$

Modifications to the main algorithm

To support learning from partial solutions, we modify Algorithm. 1 as follows to enable buffering and sampling from partial solutions. The fully updated algorithm is shown in § 3.4.6.

Guided-Sampling: In § 3.3.1, we mentioned that full solutions are sampled for each question x as $\hat{y} \sim P_\theta(\hat{y}|x)$. With PCS prefixes, compared with sampling a solution from scratch, generating solutions with these prefixes reduces the generation length thus the model can more efficiently explore the solution space. This guided sampling process is described in more detail in Algorithm. 2. Note that since the empty solution y^0 is in the buffer $\mathcal{B}$ since initialization, therefore model can still generate and explore the space from scratch and not always follows the existing solution prefixes.

Identify partially-correct prefixes: As mentioned in § 3.3.3, if a solution $\hat{y}$ does not produce the expected result z^* but its prefix $\hat{y}_{\leq i}$ is partially-correct, the model can still learn from its prefix. However, an important task here is to identify the longest partially-correct prefix for learning, in other words, locate the exact step that the solution deviates from a correct reasoning path. We can achieve this simply by backtracking the intermediate states and find the first state that is equivalent to any of the states from a saved solution. [4]

Filtering solution prefixes: With the inclusion of partially-correct solutions, we need to slightly change the two filtering criteria in § 3.3.1. For deduplication, while we still use AST to rule out changes with non-semantic tokens such as white space, we also check if the

4. In practice, we use a `state` → `solution prefix` dictionary and the lookup takes a negligible amount of time.

partially-correct solution prefix $\hat{y}_{\leq i}$ is a prefix of another known PCS in $\mathcal{B}$. For the same reason, when saving a new partially-correct solution $\hat{y}$, we need to prune out any existing solution in $\mathcal{B}$ that is a prefix of $\hat{y}$. As for the length constraint, the same principle still applies, but now it is compared against other partially-correct solution that *executes to the same state*.

Learning objective: As partially-correct solutions are solution prefixes $y_{\leq i}$ missing the later part $y_{>i}$, with an auto-regressive generation model, the learning of $P_\theta(y_{\leq i}|x)$ is independent of $y_{>i}$. Thus the learning objectives in § 3.3.2 do not need to change with the inclusion of PCS in the buffer for learning. The only difference is that the end-of-sequence "`⟨eos⟩`" token is not appended to the PCS as those solutions are not yet finished.

3.4 Experiments

3.4.1 Experimental Setup

Datasets. We evaluate on two math reasoning datasets, in which we generate straight-line Python programs as solutions to solve math problems described in natural language. We finetune the LMs to output such program solutions using only the natural language problem description as the input.

▷ **MathQA-Python-Filtered:** The original MathQA-Python consists of 19.2K training examples of NL and Python program pairs (Austin et al., 2021b). However, we find the raw dataset to contain many questions that share the same question templates and only differ in concrete number across the train/dev/test sets. To better understand the generalization of the trained models, we derive a deduplicated version of the dataset by first merging the train and dev data and then perform template-based deduplication. Partly inspired by (Finegan-Dollak et al., 2018), we re-split the train and dev set based on the question templates, resulting in 6.8K/0.7K train/dev data for the filtered version.[5] While we mainly experiment on the filtered version, we report performance on both versions when compared with previous methods.

5. We will release the processing scripts for replication and comparison.

▷ **GSM5.5K-Python:** The grade-school-math (GSM8K) dataset (Cobbe et al., 2021) contains 7.5K training data points. Since it only provides natural language solutions with math formulas and does not have a dev set, we first reserved 20% of the training data as dev set, then automatically converted the formulas to program solutions in the same style as MathQA-Python. As the result, we finetune our models with the 5.5K successfully converted training examples. Note that the natural language solutions/explanations are not used as input to the models in our experiments.

Evaluation metrics: Following recent work in neural program synthesis (Austin et al., 2021b; Chen et al., 2021b; Chowdhery et al., 2022) and math reasoning (Cobbe et al., 2021), we use PASS@k as our main evaluation metric. It allows the model to sample k solutions for each question and the task is considered solved if any one of the k solutions is correct, so PASS@k can also be seen as the fraction of problems in the test/dev set being solved given k attempts. More details (*e.g.,* temperature) can be found in § 3.4.2.

Model training: We use GPT-Neo (Black et al., 2021) as our pretrained language model (LM) and mainly study two model sizes, 125M and 2.7B.[6] Following previous work (Austin et al., 2021b), we evaluate all PASS@k on the same model checkpoint that has the best PASS@1 score, but note that it might not be the best checkpoint for other k values (more discussion in § 3.4.8). Detailed hyperparameter settings can also be found in § 3.4.2.

3.4.2 Experiment Setting Details

Hyperparameters. All hyperparameters for training is shown in Table. 3.2. We use $\beta = 0.25$ in the experiments with β-MML, as a result of enumeration search among the values of $\{0.1, 0.25, 0.5, 0.9\}$. We use the default AdamW optimizer settings and slightly tuned the learning rate by trying out several values between 1.0e-3 and 1.0e-5. The difference in floating point precision is to fit the GPT-Neo 2.7B model into the memory of the GPUs. All experiments are conducted on V100-32GB GPUs.

6. We choose GPT-Neo because it was the only public LM that have been pretrained on code when we conduct the experiments.

Name	MathQA	GSM8K.
# Training Steps	50K	25K
Learning Rate (LR)	1.0e-4	
Optimizer	AdamW	
Adam Betas	(0.9, 0.999)	
Adam Eps	1.0e-8	
Weight Decay	0.1	
LR Scheduler	Linear w/ Warmup	
# LR Warm-up Steps	100	
Effective Batch Size	32	
FP Precision	FP 32 for 125M, FP16 for 2.7B	
Gradient Clipping	1.0	

Table 3.2: The hyperparameters used for model training on two different types of datasets.

pass@k evaluation. We use temperature sampling and sample n solutions with $T = 0.8$, where $n = 80$ for MathQA and $n = 100$ for GSM to evaluate PASS@n, to be maximally consistent with previous work (Austin et al., 2021b; Chowdhery et al., 2022; Cobbe et al., 2021). We also report PASS@$\{5, 10, 20, 50\}$ using the n samples and the unbiased estimator proposed in (Chen et al., 2021b). We use $T = 0.2$ to sample 1 solution per specification and evaluate PASS@1.

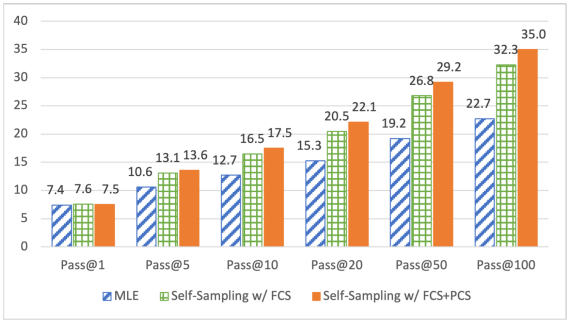
Codex few-shot settings. We estimate the Codex (Chen et al., 2021b) performance under the few-shot settings. More specifically, the prompt consists of a natural language task description "`# Generate Python code to solve the following math word problems:`" and four examples, following previous work (Chowdhery et al., 2022). Each example consists of the NL specification as a one-line comment and the gold program solutions. We evaluate PASS@k for Codex using the same sampling methods as above.

Details for self-sampling. During a training step, we sample one solution[7] for each task (*i.e.,* natural language problem) in the batch, *i.e.,* $|\hat{Y}| = 1$ in Algorithm. 1 and Algorithm. 2. Thus for each gradient update, we first compute the loss for each task based on the saved solutions in the buffer and loss functions described in Table. 3.1, then it is averaged across the 32 tasks in the batch. Note that the total number of samples we generate per task throughout training is also scaled up by the number of training epochs, which is 235 for

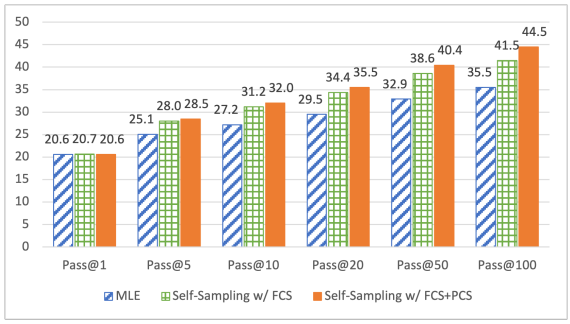
7. We also experiment with higher sampling budgets but do not observe significant improvements.

MathQA-Python-Filtered, 83 for MathQA-Python and 145 for GSM5.5K-Python. For sampling temperature, we use the same setting as inference time, with $T = 0.8$.

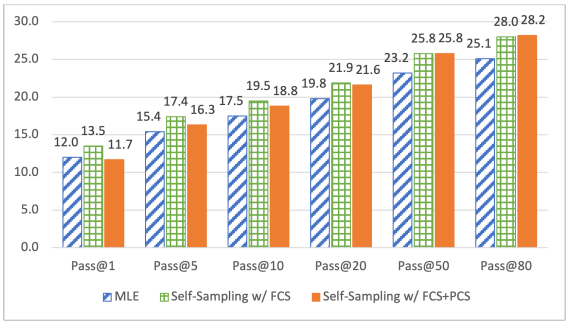
3.4.3 Main Results



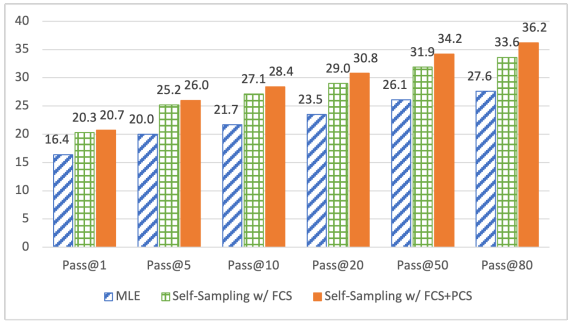
(a) GSM5.5K-Python w/ GPT-Neo 125M



(b) GSM5.5K-Python w/ GPT-Neo 2.7B



(c) MathQA-Python-Filtered w/ GPT-Neo 125M



(d) MathQA-Python-Filtered w/ GPT-Neo 2.7B

Figure 3.2: Percentage of the problems solved (PASS@k) on the dev set of GSM5.5K-Python and MathQA-Python-Filtered, comparing our self-sampling approach and the common MLE objective. All our methods include partially-correct solutions and use the MLE-Aug loss for learning.

Learning from self-sampled solutions improves pass@k. Figure. 3.2 shows the performance on the two datasets by learning from self-sampled FCSs and PCSs using MLE-Aug (orange bars), compared with MLE on single reference solution (blue bars). We can see that our proposed method can greatly improve PASS@k, especially for higher k values. By comparing different LM sizes, we can see that learning from self-sampled solutions can help with both small and large LMs, with a +12.3% and +9.0% PASS@100 improvement on GSM5.5K-Python for GPT-Neo-125M and GPT-Neo-2.7B, respectively and a +3.1% and +8.6% PASS@80 improvement on MathQA-Python-Filtered for GPT-Neo-125M and GPT-Neo-2.7B, respectively. We note that our approach does not improve PASS@1, which

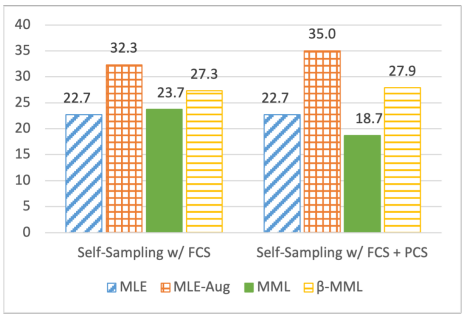


Figure 3.3: PASS@100 comparison of various loss functions (§ 3.3.2) under different self-sampling strategies. Results are on the dev set of GSM5.5K-Python with finetuned GPT-Neo 125M model. $\beta = 0.25$ for β-MML. Full results available as Table. 3.5 in § 3.4.5.

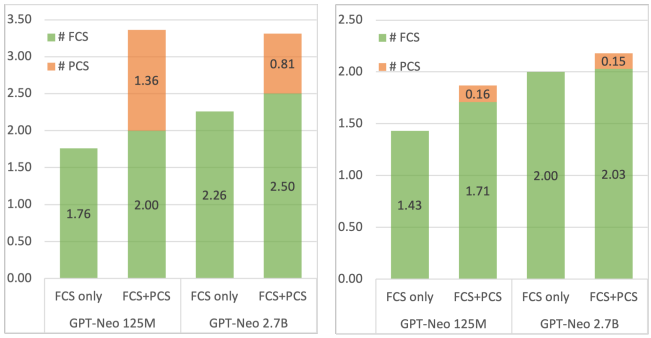


Figure 3.4: Number of saved FCSs and PCSs per problem for GSM5.5K-Python (left) and MathQA-Python-Filtered (right), with different self-sampling strategies and model sizes. # FCSs *includes* the reference solution.

is expected as learning from multiple targets mainly helps with increasing the diversity of the sampled solutions rather than improving the most-probable solution (for which MLE is better suited).

Partially-correct solutions improve model performance. We next show the effects of including partially-correct solutions on PASS@k performance in Figure. 3.2 (green bars vs orange bars) and the number of saved FCSs and PCSs in Figure. 3.4. First, we observe from Figure. 3.4 that using partial correctness not only results in PCSs being saved and directly learned from, but it also boosts the number of FCSs being found with the guided-sampling process. As a result, most PASS@k performances drop if we do not include partially-correct solutions in the buffer, as the model learns from a smaller number of FCSs and PCSs as targets. The one exception is the GPT-Neo 125M model on the MathQA-Python-Filtered dataset, where we do not observe any advantage/disadvantage of using PCSs.

MLE-Aug loss function works the best. We next study the effects of different objective functions for learning from multiple targets as described in § 3.3.2. We also experiment under different self-sampling strategies (*i.e.,* FCS only or FCS + PCS), and our experiment results on GSM5.5K-Python with the GPT-Neo 125M model are shown in Table. 3.5. We can see that MLE-Aug loss results in the biggest improvement compared to other losses both with just FCSs and with FCSs + PCSs. MML performs the worst: it

only marginally improves over MLE with only FCS and performs worse than MLE when also learning from PCSs. As discussed in § 3.3.2 and Table. 3.1, the gradient of MML is in proportional to the likelihood given by the model, thus it encourages the model to put all weight on one solution in the buffer. As MLE already learns from the gold reference solution, it is hard for MML to make improvements with self-sampled solutions, and the performance may even decrease when MML puts all weight on an incomplete partially-correct solution. In contrast, the gradients of MLE-Aug objective are equally distributed among the targets, which leads to more diversity in its generation due to a more balanced source of learning signals. β-MML loss is proposed to alleviate the aforementioned issue for MML loss, but we do not observe an advantage of using it instead of the MLE-Aug loss in our experiments.

3.4.4 Additional Analysis

Diversity of the solutions. By inspecting the k generated solutions for each task, we find that there is more diversity in the solutions that the model generates using our method. More specifically, we calculate the ratio of *unique* solutions from the 100 samples for the comparison in Figure. 3.2a, and find that 30.5% of them are unique for our approach but only 20.8% for the model trained with MLE.

Dynamics between # of PCSs and FCSs saved in the buffer. As discussed above, more saved solutions typically results in better PASS@k performance. Interestingly, when comparing different LM sizes, we can see that while the sum of partially and fully-correct solutions sampled and saved in the buffer are about the same (*i.e.,* 3.36 and 3.31) for GSM5.5K-Python dataset in Figure. 3.4, around 60% of them are FCS for the small model while it is 76% for the larger model. The difference in percentage of PCSs left in the buffer also reflects the model's ability for completing partially-correct solution prefixes. We also find that during early stages of training, the number of PCSs rapidly grows while the model is relatively weak to sample FCSs, thus the PCSs help enriching the learning signal and preventing overfitting early-on. More discussions about this can be found in § 3.4.8.

Models	Original Version		Filtered Version	
	PASS@1	PASS@80	PASS@1	PASS@80
Previous work:				
Codex Davinci† (Chen et al., 2021b)	6.0	42.0	5.0	**40.0**
LaMDA 68B* (Austin et al., 2021b)	-	79.5	-	-
LaMDA 137B* (Austin et al., 2021b)	-	81.2	-	-
Ours:				
GPT-Neo 125M w/ self-sampling FCS + PCS	**77.6**	**84.7**	11.7	28.2
GPT-Neo 2.7B w/ self-sampling FCS + PCS	-	-	**20.7**	36.2

Table 3.3: Comparison with previous methods on the original (test set used) and filtered version (dev set used) of the MathQA-Python dataset. *: model not pretrained on code. †: few-shot learning results. -: no results available.

Comparison to previous works Here we compare with previous work on both the original and the filtered versions of MathQA-Python datasets in Table. 3.3. On the original dataset, self-sampling with GPT-Neo 125M is able to outperform previous methods that finetune 137B model pretrained on natural language. We also compare with Codex model used in a few-shot setting (more details in § 3.4.2), and find that on the much harder filtered dataset, a 2.7B GPT-Neo model finetuned with our methods obtains much better PASS@1 but with lower PASS@80. By inspecting the output from Codex, we discover that its outputs are much more diverse than finetuned models, which contributes to a higher PASS@80 even under the few-shot setting. Comparison with previous work on the GSM dataset is in § 3.4.5 due to limited space.

3.4.5 Additional Experiment Results

Comparing GSM performance with previous work. Here we compare our method with previous work on the original test sets of GSM8K. The results are shown as Table. 3.4. On GSM8K, some of the prior works are evaluated on a different format of NL inputs than ours, so they are not directly comparable, but we still include them to help better position the performance of our methods. We test Codex using the same input in a few-shot setting, and we find that similar with the result on MathQA in Table. 3.3, our method achieves better PASS@1 while being significantly worse in PASS@100 compared with Codex. We hypothesize that as Codex model is used tested few-shot setting and not finetuned, it does not suffer from the overfitting issue we mentioned. This leads to great diversity but poor accuracy

Models	PASS@1	PASS@100
Previous work:		
OpenAI 6B*♣ (Cobbe et al., 2021)	21.8	70.9
PaLM-Coder 540B†♣ (Chowdhery et al., 2022)	**50.9**	-
LaMDA 137B*†♣ (Chowdhery et al., 2022)	7.6	-
Codex Cushman† (Chen et al., 2021b)	5.0	58.0
Codex Davinci† (Chen et al., 2021b)	17.0	**71.0**
Ours:		
GPT-Neo 2.7B w/ self-sampling FCS + PCS	19.5	41.4

Table 3.4: Compare with previous methods on the original test set of GSM8K dataset. *: model not pretrained on code. †: few-shot learning results. ♣: different setting from ours[8].

		# Sols. in $\mathcal{B}$		pass@k(%)					
Self-Sampling	Loss Func.	FCS	PCS	k=1	k=5	k=10	k=20	k=50	k=100
-	MLE	-	-	7.4	10.6	12.7	15.3	19.2	22.7
FCS only	MML	1.48	-	6.9	11.0	13.3	16.0	20.1	23.7
	MLE-Aug	**1.76**	**-**	**7.6**	**13.1**	**16.5**	**20.5**	**26.8**	**32.3**
	β-MML	1.57	-	7.5	11.7	14.5	17.9	23.1	27.3
FCS + PCS	MML	1.40	1.10	5.5	9.0	11.0	13.1	16.2	18.7
	MLE-Aug	**2.00**	**1.36**	**7.5**	**13.6**	**17.5**	**22.1**	**29.2**	**35.0**
	β-MML	1.62	1.14	7.2	12.0	14.9	18.4	23.6	27.9

Table 3.5: Full comparison of various loss functions (§ 3.3.2) with different self-sampling strategies. Results are on the dev set of GSM5.5K-Python with GPT-Neo 125M as the base model. Best performance within the same category is in **bold** and ones *worse than MLE* is underlined. $\beta = 0.25$ for β-MML.

during generation. However, due to the little information we have about Codex (*e.g.,* model size, training data), it is hard to derive any further conclusion.

Ablation results on loss functions. Here we show the full results on the ablation of loss functions in Table. 3.5. We can see that trends observed from PASS@100 in Figure. 3.3 are consistent with other PASS@k results, as MLE-Aug loss beats other two loss functions on all PASS@k. And using MML loss when adding PCSs for learning results in worse performance than MLE for PASS@1 as well. Moreover, from the number of FCSs and PCSs saved in the buffer $\mathcal{B}$, we can also observe that using MLE-Aug loss results in more FCSs and PCSs being saved, thus further encourages diversity in generation.

8. Natural language explanations of the solutions are used as input and the few-shot exemplars are not in the same format as ours.

Algorithm 3 Training Update with Partially Correctness

Initialize: (only once before training starts)
Solutions buffer $\mathcal{B} = \{y^0, y^*\}$ with an empty and the reference solution
Reference solution states $(s_1^*, s_2^*, ..., s_t^*)$ where $s_i^* = \mathcal{T}(y_{\leq i})$
State-prefixes mapping $\mathcal{M} = \{s_i^* \rightarrow \{y_{\leq i}\}\}_{i=1}^{t}$
Input:
Parameterized model $P_\theta(y|x)$
A training example (x, y^*, z^*)
Tracing function $\mathcal{T} : \mathcal{Y} \rightarrow \mathcal{S}$ to obtain intermediate states

$\hat{Y} \leftarrow$ *SampleSolutions*$(x, P_\theta, \mathcal{B})$ /* call Algorithm. 2 */
for $\hat{y}$ **in** $\hat{Y}$ **do**
  for $i \leftarrow |\hat{y}|; i \neq 0; i \leftarrow i - 1$ **do**
    $s_i \leftarrow \mathcal{T}(\hat{y}_{\leq i})$ /* get intermediate state for each solution prefix $\hat{y}_{\leq i}$ */
    if *PartialCorrectnessCriteria*$(s_i, \mathcal{M})$ **then**
      $Y_S \leftarrow \mathcal{M}(s_i)$ /* get existing prefixes that executes to state s_i */
      if not *isDuplicate*$(\hat{y}_{\leq i}, Y_S)$ **then**
        $\mathcal{B} \leftarrow$ *updateBuffer*$(\hat{y}_{\leq i}, \mathcal{B})$
        $\mathcal{M} \leftarrow$ *updateMapping*$(\hat{y}_{\leq i}, \mathcal{M})$
      end if
      continue /* we only need the longest matching prefix */
    end if
  end for
end for
$\theta \xleftarrow{\text{update}} \nabla_\theta \mathcal{L}(x, \mathcal{B}, P_\theta)$

3.4.6 Full Learning Algorithm with Partial Correctness

Our general learning framework in shown as Algorithm. 1 and it is further extended in § 3.3.3. Here we show a complete version of the algorithm with using partially-correct solutions in Algorithm. 3. Additionally, here are the detailed explanation of the data structure and functions used in it:

▷ **Mapping** $\mathcal{M}$: This is a data structure that maps an intermediate state to a set of solution (prefixes) that execute to that state, *i.e.,* $\mathcal{M} : \mathcal{S} \rightarrow \mathcal{Y}^n$. In this mapping, we save *all* PCSs and their intermediate states, *including all prefixes* of any PCS. We use this to significantly speed up the lookup process as mentioned in § 3.3.3;

▷ **Function** ***PartialCorrectnessCriteria***$(s_i, \mathcal{M})$: Since all states for all known PCSs are saved in $\mathcal{M}$, to know whether a prefix $\hat{y}_{\leq i}$ is partially-correct, we only need to check if its state matches any of the known states for a PCS, *i.e.,* if $s_i \in \mathcal{M}$;

▷ **Function** ***isDuplicate***$(\hat{y}_{\leq i}, Y_S)$: As mentioned in § 3.3.3, we use AST and length constraint to rule out "trivial variants" and identify new PCSs to save in the buffer $\mathcal{B}$. Here the solutions to compare are the set of solutions Y_S that reaches the same intermediate state, *i.e.,* being

state-based equivalent;

▷ **Function *updateBuffer***($\hat{y}_{\leq i}, \mathcal{B}$): Here we not only need to add the new PCS into the buffer $\mathcal{B}$, but also need to prune out the saved solutions that are prefix of $\hat{y}_{\leq i}$;

▷ **Function *updateMapping***($\hat{y}_{\leq i}, \mathcal{M}$): Here we need to save the states of all prefixes of an identified partially-correct solution, thus we will loop through all prefixes of $\hat{y}_{\leq i}$ and obtain its execution state, then update $\mathcal{M}$ accordingly. As mentioned above, existing PCSs may be a prefix of the new PCS, so we also need to prune out such existing PCSs from mapping $\mathcal{M}$.

3.4.7 Qualitative Analysis

In Table. 3.6, we show more examples of the fully-correct and partially-correct solutions that the models found during self-sampling, from both the MathQA and GSM datasets. First, we can see that for some NL problems, it is possible that no FCS or PCS can be found with self-sampling, as in *MathQA-Example-1* and *MathQA-Example-1*. Take *MathQA-Example-2* as an example, the question is quite straightforward thus it leaves very little room for the existence of other correct solutions, as the reference solution is already very short. Moreover, we can also observe that the ways self-sampled FCS and PCS differ from the reference solution vary a lot. In *MathQA-Example-2*, *GSM-Example-1* and *GSM-Example-2* the sampled FCSs complete the task with very different paths compared with the reference solution, and actually result in using fewer lines of code. Another way of getting FCS or PCS is to perform small and local perturbations, *e.g.,* switch the two sides of a addition or re-order the two non-dependent statements, as shown in other examples. We find that these local perturbations are more common in general in both datasets, as such patterns are easier for the model to learn.

3.4.8 Tracking Training Progress

Learning from self-sampled solutions mitigates overfitting. Here we shown the PASS@k performance curve with respect to the training process in Figure. 3.5. From the curves, we can observe that for MLE, while PASS@1 and PASS@5 generally improves during training, other PASS@k for higher k actually decreases after reaching the peak performance in early epochs, which is consistent with previous findings (Cobbe et al., 2021). This is due

NL Problem Descriptions	Ref. Solution	Self-Sampled FCS	Self-Sampled PCS
(MathQA-Example-1): The charge for a single room at hotel P is 70 percent less than the charge for a single room at hotel R and 10 percent less than the charge for a single room at hotel G. The charge for a single room at hotel R is what percent greater than the charge for a single room at hotel G?	n0=70.0 n1=10.0 t0=100.0-n0 t1=100.0-n1 t2=t0/t1 t3=t2*100.0 t4=100.0-t3 t5=t4/t3 answer=t5*100.0	n0=70.0 n1=10.0 t0=100.0-n1 t1=100.0-n0 t2=t0/t1 t3=t2*100.0 answer=t3-100.0	-
(MathQA-Example-2): If john runs in the speed of 9 km/hr from his house, in what time will he reach the park which is 300m long from his house?	n0=9.0 n1=300.0 t0=n0*1000.0 t1=n1/t0 answer=t1*60.0	-	-
(MathQA-Example-3): A class consists of 15 biology students and 10 chemistry students. If you pick two students at the same time, what's the probability that one is maths and one is chemistry?	n0=15.0 n1=10.0 t0=n0+n1 t1=n0/t0 t2=n1/t0 t3=t0-1.0 t4=n1/t3 t5=n0/t3 t6=t1*t4 t7=t5*t2 answer=t6+t7	n0=15.0 n1=10.0 t0=n0+n1 t1=n0/t0 t2=n1/t0 t3=t0-1.0 t4=n1/t3 t5=n0/t3 t6=t1*t4 t7=t5*t2 answer=t7+t6	n0=15.0 n1=10.0 t0=n0+n1 t1=n0/t0 t2=n1/t0 t3=t0-1.0 t4=n0/t3 t5=n1/t3
(GSM-Example-1): Ellie has found an old bicycle in a field and thinks it just needs some oil to work well again. She needs 10ml of oil to fix each wheel and will need another 5ml of oil to fix the rest of the bike. How much oil does she need in total to fix the bike?	n0=2 n1=10 n2=5 t0=n0*n1 answer=t0+n2	n0=10 n1=5 t0=n0+n1 answer=n0+t0	n0=10 n1=5 n2=2
(GSM-Example-2): There is very little car traffic on Happy Street. During the week, most cars pass it on Tuesday - 25. On Monday, 20% less than on Tuesday, and on Wednesday, 2 more cars than on Monday. On Thursday and Friday, it is about 10 cars each day. On the weekend, traffic drops to 5 cars per day. How many cars travel down Happy Street from Monday through Sunday?	n0=20 n1=100 n2=25 n3=2 n4=10 t0=n0/n1*n2 t1=n2-t0 t2=t1+n3 t3=n4*n3 t4=t0*n3 answer=t3+n2 \ +t2+t3+t4	n0=25 n1=2 n2=20 n3=100 n4=10 t0=n0-n1 t1=n2/n3*n0 t2=t0-t1 t3=t2+n4 t4=n0-t3 answer=t4+n3	n0=2 n1=25 n2=20 n3=100 n4=10

Table 3.6: More examples of self-sampled fully-correct (FCS) and partially-correct solutions (PCSs). "MathQA" denotes the MathQA-Python-Filtered dataset and "GSM" denotes the GSM5.5K-Python dataset. All solutions are from the *final buffer after training* a GPT-Neo 2.7B model, while learning from self-sampled FCS+PCS with the MLE-Aug loss.

to overfitting: in the early stage of training, the model is less confident about its predictions thus the sampled k solutions are very diverse, and while training continues, it overfits to the one reference solution provided for learning thus leads to poor generalization when evaluated by PASS@k with high k values. Figure. 3.5 also shows how our proposed self-sampling method can mitigate the overfitting problem, as it keeps improving or maintaining PASS@$\{5, 10, 20\}$

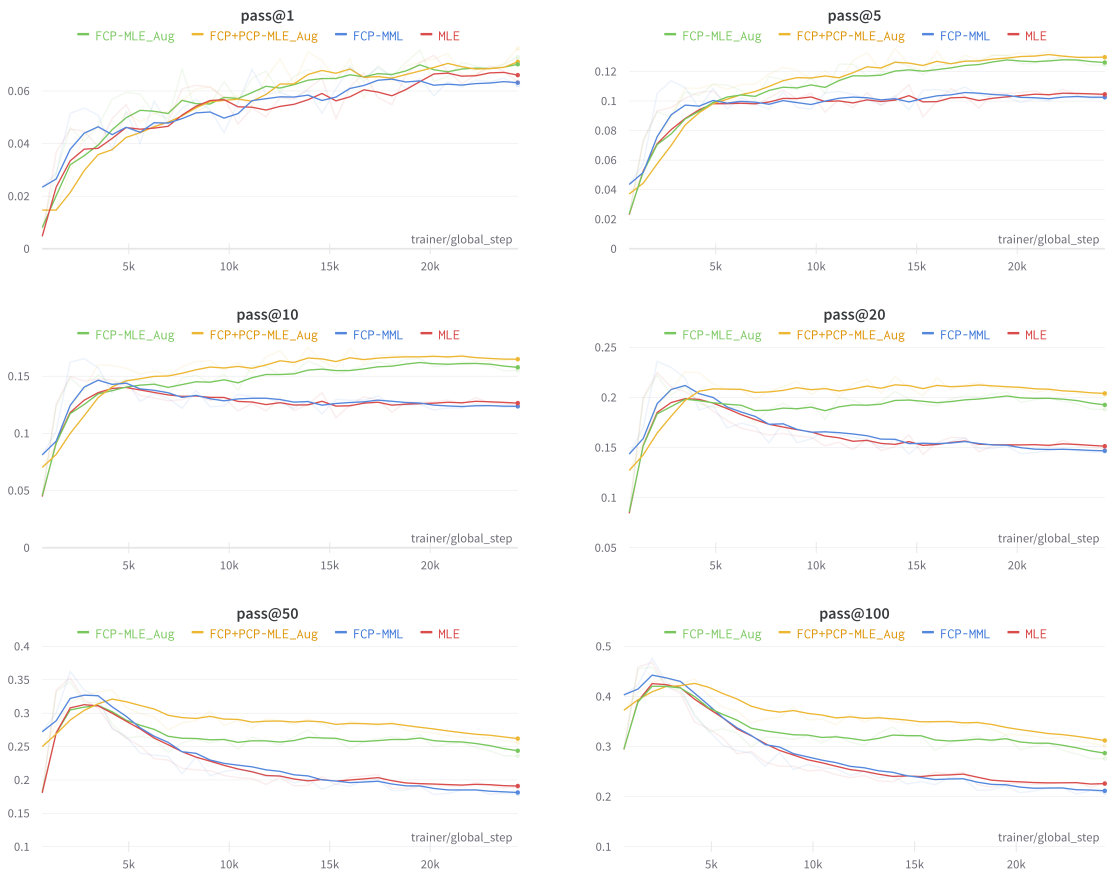
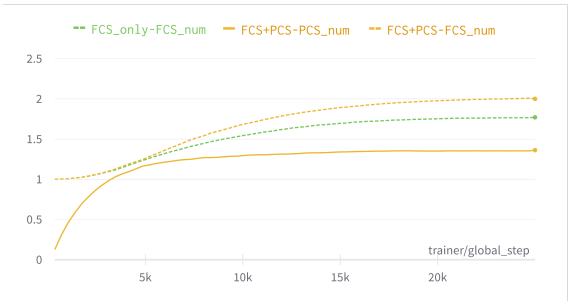


Figure 3.5: How PASS@k on the dev set evolve during training. Results shown on GSM5.5K-Python dataset with GPT-Neo 125M model. Exponential moving average smoothing is applied for more clarity, but original curve is shown in shade.

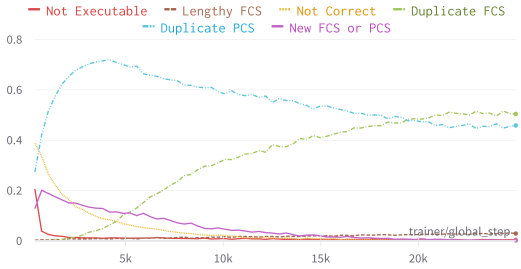
while such performances start decreasing for MLE. Though it also shows improvements for PASS@$\{50, 100\}$, but the performance still decreases in later training stages. Here we can also see the importance of suitable learning objective, as MML has almost no effect in mitigating such overfitting issue.

Early stopping is needed when prioritizing high k value for pass@k. In our experiments, we select the model checkpoint with the best PASS@1 performance to evaluate all PASS@k. This setup aims to choose the best model that can solve the task with a small number of attempts (which corresponds to smaller k value), as studied in (Austin et al., 2021b). We can also observe that with our methods, the best PASS@1 checkpoint also yields the best or close to the best PASS@$\{5, 10, 20\}$ performances. However, in certain applications where large number of attempts are allowed, PASS@k with high k values should

be prioritized. An example is to generate candidate solutions before reranking (Cobbe et al., 2021). In this case, an earlier checkpoint (*e.g.,* one with best PASS@100) should be used instead, which is not the best checkpoint for PASS@k where k is small. Also note that our proposed method are not suitable for these applications, as we observe no improvement on the peak PASS@$\{50, 100\}$ performances. We think this because when such peak performance is reached, it is still in the early stage of training thus not many FCSs or PCSs have been saved in the buffer yet.



(a) Growth of the number of saved FCS and PCS during training. # FCS includes the gold solutions.



(b) Distribution of the characterization of self-sampled solutions during training.

Figure 3.6: How self-sampling evolves throughout the training process. Results shown as training the GPT-Neo 125M model on the GSM5.5K-Python dataset with MLE-Aug loss.

Partially-correct solutions help in early training stages. To show how self-sampling effects training, in Figure. 3.6a we show how the size of the buffer progresses during training. From the curves, we can see that in the early training stages (*i.e.,* first 5k steps), the number of saved PCSs rapidly grows while the number of FCSs only slightly increases. In later stages of training, the growth of buffer size is mainly contributed by more FCSs being sampled and saved while the number of PCSs stays steady. Also when compared to learning only with FCSs, learning with FCSs + PCSs eventually accumulates more FCSs in the buffer (green dotted line vs yellow dotted line). In addition, we show how the distribution of the outcomes of self-sampled solutions changes throughout training in Figure. 3.6b. We can see that in the early training stages, the ratio of not executable/incorrect solutions quickly drops to almost zero. At the same time, the ratio of new FCS or PCS being saved reaches the peak. As training proceeds, the models are mostly sampling known FCS or PCS as the size of the buffer converges as well. But the number of self-sampled fully-correct solutions

gradually overtakes the partially-correct ones.

3.5 Limitations and Future Work

More general definition of (partial) correctness. In this chapter, we define partial correctness based on state-based solution equivalence. This is a conservative way for defining solution equivalence as it requires exact match of the sets of variable values, but a solution could be partially correct and yet, not have an exact match of variable values because some of these values may not needed for future computation. In the future, we want to explore ways to relax this restriction that will help us find more partially correct solutions in an efficient manner. Besides, our partial correctness definition requires the existence of at least one fully-correct solution and when such reference solution is not available from the dataset (*i.e.,* in a weakly-supervised setting), we would need to first sample an FCS that matches the gold execution result to begin with. In addition, we simply use the matching of execution results to define correctness, which is susceptible to spurious solutions that achieves the correct result by coincidence. For math reasoning, we find such spurious solutions to be quite rare[9], as the correct answer is typically numeric which is less likely for a semantically wrong solution to obtain the correct answer by chance. But methods as (Chen et al., 2022a; Zhong et al., 2020) may be needed for this definition of correctness to be more robust on other domains.

Towards generating general programs. While we focus on the domain of generating solutions for math reasoning in this chapter, here we reflect on how our method can be applied to program synthesis in general. However, general programs might contain complex structures such as conditions (*e.g.,* `if-else`) or loops (*e.g.,* `while-do`) as opposed to straight-line programs in the math-reasoning domain. Dealing with these complex structures poses additional challenges because most neural program synthesis models perform left-to-right auto-regressive generation, and the changes to the control flow break the alignment between program generation and program execution (Chen et al., 2018a, 2021d; Nye et al., 2021).

9. We manually inspected the self-sampled FCSs by GPT-Neo 2.7B on 100 tasks of GSM5.5K and found spurious solutions only exist for 3 of them.

There are two potential ways to extend our technique to address the problem. First, we can treat a branch or a loop as an atomic unit (*i.e.,* a block whose state is the state after executing all statements within it), then we can apply state-based equivalence in the same way. Second, because our technique only requires execution after the full programs are generated, we can still evaluate and compare program states based on intermediate states.

3.6 Related Work

Weakly-supervised semantic parsing. Many previous work in learning semantic parsers from weak supervision follows the same process of sampling programs and maximizing the probability of the correct ones (Guu et al., 2017b; Krishnamurthy et al., 2017; Min et al., 2019; Ni et al., 2020). Our work differs as our tasks contain one reference solution for each task as opposed to only the final answer like weakly-supervised semantic parsing tasks. Thus, our work leverages the reference solution for sampling and defines partial correctness based on known solutions. Because of the problem setup difference, we found that the conclusions in (Guu et al., 2017b) about loss functions do not generalize to our case.

Execution-guided code generation. Our work relates to execution-guided code generation as we leverage intermediate states of math solutions to guide the sampling process. In code generation literature, intermediate program execution states are used to prune the search space (Li et al., 2022c; Liang et al., 2017; Wang et al., 2018c) or condition further generation on the execution states(Chen et al., 2018a, 2021d; Ellis et al., 2019; Nye et al., 2020a, 2021). The key difference of these methods from ours is that they require doing both decoding and execution at inference time, while our work only uses execution during training, which reduces decoding overhead.

Learning from partial reward for program synthesis. There are parallels between multi-target learning and the reinforcement learning setting with sparse rewards for generating programs (Agarwal et al., 2019; Bunel et al., 2018; Liang et al., 2017, 2018; Simmons-Edler et al., 2018). Similarly, our approach of identifying partial correctness of solutions is similar to partial rewards. But instead of discounting an entire trajectory with a low reward as in RL, we truncate the solution to a partially-correct prefix and assign it the "full reward",

which is a main contribution of this work.

3.7 Summary

In this chapter, we study how to better improve language models for math reasoning during supervised finetuning (SFT) and propose to let pretrained language models sample additional solutions for each problem and learn from the self-sampled solutions that are correct or partially-correct. We define partial correctness by tracing and matching intermediate execution states. We experiment on different math reasoning tasks and show that such partially-correct solutions can help more efficient exploration of the solution space and provide useful learning signal, which improves the PASS@k performance. Overall, our proposed method can improve PASS@k from 3.1% to 12.3% compared to learning from a single solution with MLE.

Chapter 4

Teaching Language Models to Reason about Code Execution

In this chapter, we present Naturalized Execution Tuning (NExT), an iterative, self-training method that teaches LLMs to reason about code execution in natural language by inspecting the execution traces of the programs. To facilitate the understanding of execution traces (a sequence of execution states) of programs, we first develop an LLM-friendly trace representation. Given the programs and their execution traces, we then prompt the LLMs to reason about program execution in chain-of-thought style, step-by-step rationales in natural language. NExT trains the LLMs on the rationales that lead to correct program outputs and the program outputs themselves. Experiments on two program repair datasets show that NExT significantly improves the quality of the natural language rationales, as well as the program fix rates.[1]

4.1 Introduction

Recent years have witnessed the burgeoning of large language models (LLMs) trained on code (Anil et al., 2023; Austin et al., 2021b; Chen et al., 2021b; Li et al., 2023; Roziere et al., 2023; Touvron et al., 2023a). While those LLMs achieve impressive performance in assisting

1. The content of this chapter is directly adapted from *"NExT: Teaching Large Language Models to Reason about Code Execution"* by Ni *et. al* , 2024.

developers with writing (Chen et al., 2021b), editing (Fakhoury et al., 2023), explaining (Hu et al., 2018), and reviewing (Li et al., 2022d) code, they still struggle on more complex software engineering tasks that require reasoning about the runtime execution behavior of programs (Ma et al., 2023). On the other hand, it is not always sufficient for the model to suggest good code solutions, but it is often necessary to provide an explanation to developers to document what the change does and why it is needed. These explanations can help developers better understand the code solutions from models and make more informative decisions. (Cito et al., 2022; Kang et al., 2023; Ross et al., 2023).

For example, *program repair* (Chen et al., 2018b; Le Goues et al., 2019; Li et al., 2020) is the task of fixing bugs in a program. Human developers usually learn to debug and fix code by interacting with code interpreters or debuggers to inspect the variable states of executed lines (Siegmund et al., 2014). Such practice helps them acquire a *mental model* of program execution (Heinonen et al., 2022), so that they could mentally simulate code execution in a more abstract manner using natural language as in rubber duck debugging (Hunt and Thomas, 1999). Therefore, a program repair model would be more helpful to developers if the model could carry out similar reasoning about program execution in order to explain bugs to programmers.

With this inspiration, our goal is to improve the ability of LLMs to reason about program execution when solving coding tasks. In this chapter we propose Naturalized Execution Tuning (NExT), which aims to teach LLMs to reason with code execution by inspecting program execution traces and reasoning about the code's runtime behavior in natural language (NL). At a general level, for a coding task, the main idea is to train a model to generate intermediate NL rationales, as in chain-of-thought reasoning (Wei et al., 2022b), but to provide the model with a trace of the execution of the program in question, so the rationale can be more accurate and grounded on program semantics. Teaching LLMs to reason about program execution in NL would not only offer better interpretability, it could also increase the diversity of solutions predicted by the model (Yin et al., 2023).

Fig. 4.1 illustrates our proposed approach when applied to program repair. Given an NL task instruction (x in Fig. 4.1) and a buggy program ($\tilde{y}$), as well as the execution traces of the program (ϵ), an LLM solves the task (*e.g.,* predict the fixed code $\hat{y}$) using

chain-of-thought (CoT) reasoning to generate a *natural language rationale* ($\hat{r}$) leveraging the execution information[2]. Intuitively, program traces encode useful debugging information such as line-by-line variable states (*e.g.,* the value of `str_list` in ϵ, Fig. 4.1) or any exceptions thrown, which could be useful for LLMs to identify and fix bugs by reasoning over the expected and the actual execution results (*e.g.,* ***"highlighted text"*** in $\hat{r}$). To help LLMs understand execution traces, NExT represent traces as compact inline code comments (*e.g.,* `# (1) str_list=...` in ϵ, more in §4.3), without interrupting the original program structure.

While execution traces capture informative runtime behavior, we find it challenging for LLMs to effectively leverage them out-of-box through CoT prompting (§4.3). Therefore we opt to finetune LLMs on high-quality CoT rationales that reason about program execution (§4.4). NExT uses weakly-supervised self-training (Zelikman et al., 2022) to bootstrap a synthetic training set by sampling rationales that lead to correct task solutions (*e.g.,* fixed code $\hat{y}$ in Fig. 4.1) verified by unit tests (Ye et al., 2022). Using unit tests as weak supervision, NExT learns to discover task-specific, execution-aware NL rationales without relying on laborious manual annotation of rationales (Chung et al., 2022; Lightman et al., 2023; Longpre et al., 2023) or distilling such data from stronger teacher models (Fu et al., 2023; Gunasekar et al., 2023; Mitra et al., 2023; Mukherjee et al., 2023). NExT executes this self-training loop for multiple iterations (Anthony et al., 2017; Dasigi et al., 2019b), solving more challenging tasks with improved success rate and rationale quality (§4.5).

We evaluate NExT with the PaLM 2-L model (Anil et al., 2023) on two Python program repair tasks. Experiments (§4.5) show that NExT significantly improves PaLM 2's ability to reason about program execution in natural language, improving the program fix rate on Mbpp-R by 26.1% and HumanEvalFix-Plus by 14.3% absolute, respectively. When compared against a strong self-training program repair approach without predicting NL rationales (Ye et al., 2022), our model achieves comparable accuracy with significantly improved sample diversity. Interestingly, while our model learns to reason with pre-existing execution information in input program traces, it also generalizes to the out-of-distribution

2. While there are a variety types of execution information that we may provide to an LLM (*e.g.,* variable read/write, runtime environments), in this work we limit the execution information to program states and variable values from the execution trace, which is common information that (human) developers also use.

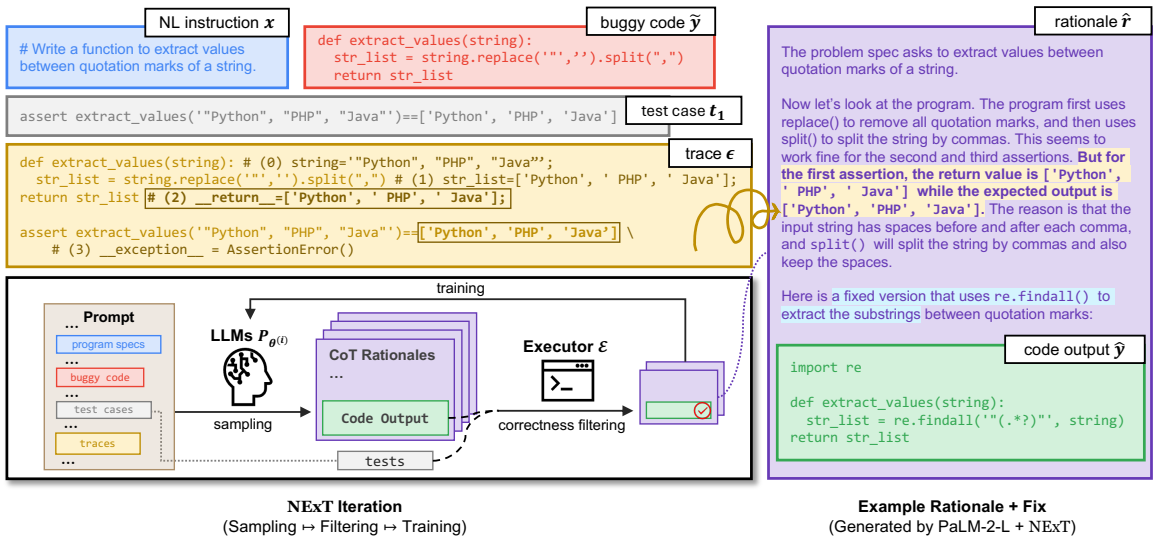


Figure 4.1: NExT finetunes an LLM to *naturalize* execution traces into the chain-of-thought rationales for solving coding tasks. It performs *iterative self-training* from weak supervision, by learning from samples that lead to correct task solutions.

```
1  def separate_odd_and_even(lst):  # (0) lst=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
2      odd_list = []  # (1) odd_list=[];
3      even_list = []  # (2) even_list=[];
4      for n in lst:  # (3) n=1; (5) n=2; (7) n=3; ...; (21) n=10;
5          if n even_list.append(n)  # (4) even_list=[1]; (8) even_list=[1, 3]; ...; (20) even_list=[1, 3, 5, 7, 9];
6          else:
7              odd_list.append(n)  # (6) odd_list=[2]; (10) odd_list=[2, 4]; ...; (22) odd_list=[2, 4, 6, 8, 10];
8      return odd_list, even_list  # (23) __return__=([2, 4, 6, 8, 10], [1, 3, 5, 7, 9])
9
10 separate_odd_and_even([1,2,3,4,5,6,7,8,9,10]) == [1,3,5,7,9], [2,4,6,8,10]
```

Figure 4.2: NExT represents execution trace as `inline comments`. More details in §4.2 and §4.6.1.

scenario where execution traces are not available at test-time. Finally, to measure the quality of model-generated rationales, we propose a *proxy-based* evaluation approach, which approximates rationale quality using the performance of smaller LLMs when prompted to solve the original task following those rationales from our models. Through both proxy-based evaluation and human annotation, we demonstrate that NExT produces helpful NL rationales which explain the causes of bugs while suggesting potential fixes. The generated rationales are of significantly higher quality compared to those from the base PaLM 2-L model.

4.2 Task: Program Repair with Traces

Here we introduce our task of program repair with execution traces using chain-of-thought reasoning.

Program Repair with Execution Traces. As in Fig. 4.1, given an instruction x and a buggy code solution $\tilde{y}$, automated program repair (Le Goues et al., 2019) aims to generate a fixed program $\hat{y}$ such that $\hat{y}$ passes all test cases $t \in T$ in an executor $\mathcal{E}$, *i.e.,* $\mathcal{E}(\hat{y}, T) = 1$ while $\mathcal{E}(\tilde{y}, T) = 0$. In this chapter, we focus on the task of program repair using execution traces (Bouzenia et al., 2023). Specifically, a **program trace** ϵ is a sequence of intermediate variable states after executing each statement in $\tilde{y}$ against a test case t. Intuitively, traces record the computation of a program, and can provide useful debugging information (*e.g.,* exceptions) to repair $\tilde{y}$.

To use LLMs to repair programs with traces, we concatenate the task instruction, the buggy code, the test cases, and their execution traces as a prompt (Fig. 4.1). To help LLMs understand program traces, we design a prompt-friendly trace representation by formatting ϵ as compact inline code comments (*i.e.,* ϵ in Fig. 4.1), as discussed later.

CoT Reasoning with Execution. We focus on using chain-of-thought reasoning (Wei et al., 2022c) to solve program repair problems by reasoning with execution, where an LLM is prompted to generate an NL rationale $\hat{r}$ together with a fixed program $\hat{y}$ as in Fig. 4.1. Specifically, we consider rationales that contain reasoning steps to identify and explain bugs in the original code (*e.g.,* the second paragraph in $\hat{r}$, Fig. 4.1), as well as suggestions to fix the buggy code (*e.g.,* *"a fixed version that uses* `re.findall()`*"* in $\hat{r}$). Since rationales are generated using traces, they often include useful reasoning about program execution that helps localize the bug, such as identifying a counterfactual between the expected and the actual variable values of a statement (*e.g.,* the *"highlighted text"* in $\hat{r}$). Such explanations can be helpful for developers to understand bugs in the original code and the model's fixed solutions (Kang et al., 2023). We therefore aim to improve the quality of NL rationales along with the fix rate by teaching LLMs to reason with execution information.

An LLM-friendly Trace Representation. The raw execution traces collected at

Datasets	Prompting	PaLM 2-L	GPT-3.5	GPT-4	Mixtral 8x7B	DS-C 33B	S-C 15.5B	Avg.
MBPP-R	Vanilla w/ trace	27.5	41.8	62.6	16.1	23.9	13.3	30.9
	+ CoT	26.6	46.4	62.8	21.1	18.2	12.6	$31.3_{+0.4}$
	+ CoT; − trace	19.0	47.1	51.3	18.1	12.9	10.6	$26.5_{-4.8}$
HEFIX+	Vanilla w/ trace	59.1	70.1	88.4	32.9	57.3	29.3	56.2
	+ CoT	48.8	75.6	84.8	34.1	30.5	16.5	$48.4_{-7.8}$
	+ CoT; − trace	43.3	72.0	82.9	25.6	22.6	18.3	$44.1_{-4.3}$

Table 4.1: Few(3)-shot prompting repair accuracy using greedy decoding. Results worse than the previous row above them are underlined in red. "DS-C" and "S-C" denote "DeepSeek Coder" and "StarCoder", respectively.

runtime contain complete variable states for each executed statement.[3] Encoding all such information in prompts is not feasible given the context limit and computation overhead of LLMs. To address this issue and make execution information more intelligible to LLMs, we propose an *inline trace representation* format, which encodes variable states as inline comments of the traced program. Fig. 4.2 shows an example. Specifically, each inline comment only encodes changed variables after executing that line. Because statements may be invoked multiple times in non-obvious orders (*e.g.,* in loops like lines 4 to 8 in Fig. 4.2), we index the variable states based on the execution order (*e.g.,* `(3) n=1;` and `(4) even_list=[1]`), and one may reconstruct the original execution footprint by following those variable states in order. We further compress the trace information for loops by omitting the variable states in intermediate iterations (*e.g.,* "..." in lines 4, 6, and 8). Intuitively, by showing states as pseudo-comments within the original code without interrupting the program structure, our trace representation is significantly more compact than existing approaches that unroll executed lines of code and pair them with line-by-line variable states (*c.f.*, Bouzenia et al., 2023; Nye et al., 2021),[4] while allowing an LLM to leverage its learned code representation to understand the additional execution effect of each statement. Implementation details about handling complex control structures are discussed in §4.6.1.

3. We use the `sys.settrace()` hook in Python.

4. As a comparison, 95% examples in our MBPP-R benchmark can fit into a 2K context window using our inline representation, while only 60% of them can fit into the same window using the Scratchpad trace format in Nye et al. (2021). A more detailed comparison is shown in Tab. 4.7.

4.3 Preliminary Study: Can LLMs reason with program traces in natural language?

Before introducing NExT, we first conduct a preliminary study to explore whether LLMs could reason with execution traces in natural language out-of-box without additional training. Answering this question will motivate our finetuning approach to improve such reasoning skills. Specifically, we follow the trace representation in §4.2 and few-shot prompt an LLM to solve program repair tasks using CoT reasoning.

Models. We evaluate the following general-purpose models: PaLM 2 (Anil et al., 2023), GPT (OpenAI, 2023)[5], and Mixtral (Jiang et al., 2024a). We also test two code-specific LLMs: StarCoder (Li et al., 2023) and DeepSeek Coder (Guo et al., 2024). Tab. 4.1 reports the results on two Python program repair datasets (see §4.5 for details).

LLMs struggle on CoT reasoning with traces. We observed mixed results when comparing vanilla prompting with traces without CoT (**Vanilla w/ trace** in Tab. 4.1) and CoT prompting with rationales (**+CoT**). Surprisingly, CoT prompting is even worse on HumanEvalFix-Plus, with an average drop of −7.8% compared to vanilla prompting, especially for code-specific LLMs (57.3 $\mapsto$ 30.5 for DeepSeek Coder and 29.3 $\mapsto$ 16.5 for StarCoder). After inspecting sampled rationales predicted by PaLM 2-L, we observe that the model is subject to strong hallucination issues, such as mentioning exceptions not reflected in the given traces. Indeed, as we later show in §4.5.2, the overall correctness rate of explaining errors in input programs among these sampled rationales from PaLM 2-L is only around 30%. Moreover, CoT reasoning is even more challenging for those models when we remove execution traces from the inputs (**+CoT;−trace**), resulting in an average performance drop of 4.8% on Mbpp-R and 4.3% on HumanEvalFix-Plus. These results suggest that while our trace representation is useful for LLMs to understand and leverage execution information for program repair (since "−trace" leads to worse results), they could still fall short on CoT reasoning using natural language with those program traces. This finding therefore motivates us to improve LLMs in reasoning with execution through finetuning,

5. We use `gpt-3.5-turbo-1106` and `gpt-4-1106-preview`.

which we elaborate in §4.4.

4.4 NExT: Naturalized Execution Tuning

We present NExT, a self-training method to finetune LLMs to reason with program execution using synthetic rationales.

Overview of NExT. Fig. 4.1 illustrates NExT, with its algorithm detailed in Algo. 4. NExT is based on existing self-trained reasoning approaches (Uesato et al., 2022; Zelikman et al., 2022), which employ expert iteration to improve a base LLM using synthetic rationales sampled from the model. Given a training set $\mathcal{D}$ of repair tasks with execution traces, NExT first samples candidate NL rationales and fixed code solutions from the LLM. Those candidate solutions are filtered using unit test execution diagnostics, and those that pass all test cases are then used to update the model via finetuning. This sample-filter-train loop is performed for multiple iterations, improving the model's rationales and repair success rate after each iteration.

Sampling rationales and code solutions. For each iteration i, we sample rationales $\hat{r}$ and fixes $\hat{y}$ in tandem from the current model $P_{\theta^{(i)}}$ (Line 5, Algo. 4). We use few-shot prompting (§4.3) when $i = 0$ and zero-shot prompting with trained models for later iterations. In contrast to existing self-training methods that leverage all training problems, NExT only samples candidate solutions from the subset of problems in $\mathcal{D}$ that are challenging for the base model $P_{\theta^{(0)}}$ to solve (Line 1). Specifically, given a metric $\mathcal{M}(\cdot)$, we only use problems $d \in \mathcal{D}$ if $P_{\theta^{(0)}}$'s metric on d is below a threshold m. Refer to §4.5 for more details about the $\mathcal{M}(\cdot)$ and m of our program repair task. Focusing on sampling solutions from those hard problems not only significantly reduces sampling cost, it also improves program repair accuracy, as it helps the model towards learning to solve more challenging problems. See §4.8 for a more detailed analysis.

Filtering candidate solutions. Given a candidate set of sampled NL rationales and their code fixes, NExT uses unit test execution results to identify plausible rationales that lead to correct fixes for learning (Line 6). Using test execution diagnostics as a binary reward function is natural for program repair tasks since each repair problem in our dataset

Algorithm 4 Naturalized Execution Tuning (NExT)

Input: Training set $\mathcal{D} = \{(x_j, \tilde{y}_j, T_j, \epsilon_j)\}_{j=1}^{|\mathcal{D}|}$ (§4.2); Development set $\mathcal{D}_{dev}$; Base LLM $P_{\theta^{(0)}}$; Number of iterations I; Executor $\mathcal{E}$; Evaluation metric $\mathcal{M}$ and threshold m

1: $\mathcal{D}_H \leftarrow \{d \mid d \in \mathcal{D}, \mathcal{M}(P_{\theta^{(0)}}, d) < m\}$ // Identify hard problems $\mathcal{D}_H$ with metric $\mathcal{M}(\cdot) < m$
2: **for** $i = 0$ **to** I **do**
3: $\quad \mathcal{B}^{(i)} \leftarrow \{\}$
4: $\quad$ **for** $(x_j, \tilde{y}_j, T_j, \epsilon_j)$ **in** $\mathcal{D}_H$ **do**
5: $\quad\quad S_j^{(i)} \sim P_{\theta^{(i)}}(r, y \mid x_j, \tilde{y}_j, T_j, \epsilon_j)$ // Sample rationales r and fixes y using trace ϵ_j.
6: $\quad\quad \mathcal{B}^{(i)} \leftarrow \mathcal{B}^{(i)} \cup \{(\hat{r}, \hat{y}) \mid (\hat{r}, \hat{y}) \in S_j^{(i)}, \mathcal{E}(\hat{y}, T_j) = 1\}$ // Filter w/ test cases T_j and add to $\mathcal{B}^{(i)}$.
7: $\quad$ **end for**
8: $\quad \theta^{(i+1)} \leftarrow \arg\max_\theta \mathbb{E}_{\mathcal{B}^{(i)}}[P_\theta(\hat{r}, \hat{y} \mid x, \tilde{y}, T, \epsilon)]$ // Finetune model $P_{\theta^{(0)}}$ with data in $\mathcal{B}^{(i)}$.
9: **end for**
10: $i^* \leftarrow \arg\max_i \sum_{d \sim \mathcal{D}_{dev}} \mathcal{M}(P_{\theta^{(i)}}, d)/|\mathcal{D}_{dev}|$ // Select the best checkpoint i^*

Output: model $P_{\theta^{(i^*)}}$

comes with unit tests to test the functional correctness of its proposed fixes (Ye et al., 2022). While we remark that this filtering criteria does not directly consider rationale quality, we empirically demonstrate in §4.5 that the quality of rationales improves as learning continues.[6]

Model training. After collecting a set of training examples $\mathcal{B}^{(i)}$, we finetune the model to maximize the probability of generating the target rationales and code fixes given the task input (Line 8). Following Zelikman et al. (2022), we always finetune the model from its initial checkpoint $P_{\theta^{(0)}}$ to avoid over-fitting to instances sampled from early iterations that are potentially of lower-quality.

Discussion. NExT can be seen as an instantiation of the rationale bootstrapping method proposed in Zelikman et al. (2022) (§ 3.1), which synthesizes latent rationales with correct answers for math and logical reasoning tasks. However, NExT focuses on program comprehension by reasoning with execution traces, which is critical for solving challenging coding tasks that require understanding execution information, such as program repair (§4.5). Besides, NExT models both rationales and programs (code fixes) as latent variables. Using unit test execution results as weak supervision, NExT is able to explore possible strategies to reason with execution and discover plausible rationales catered towards solving the specific downstream task. As we show in §4.9, rationales generated by NExT employ a variety of reasoning patterns to locate and explain bugs in our repair dataset. Finally, while we apply NExT to program repair, our framework is general and can be extended to other

6. The rationale and fix quality may plateau at a different iteration i.

Models	End-to-end Fix Rate				Proxy-based Evaluation			
	pass@1	pass@5	pass@10	pass@25	pass@1	pass@5	pass@10	pass@25
GPT-4; 3-shot	63.2	75.1	78.5	82.7	44.8	66.5	72.5	77.8
GPT-3.5; 3-shot	42.9	65.0	70.7	76.7	26.6	48.8	57.0	66.4
PaLM 2-L; 3-shot	23.2	45.7	54.7	65.0	22.5	43.4	51.9	61.5
PaLM 2-L+NExT *	$49.3_{+26.1}$	$68.1_{+22.4}$	$73.5_{+18.8}$	$79.4_{+14.4}$	$28.8_{+6.3}$	$49.9_{+6.5}$	$57.3_{+5.4}$	$65.5_{+4.0}$

Table 4.2: Improvements by NExT on the PaLM 2-L model (in subscripts) on Mbpp-R. GPT-3.5/4 results are for reference. *:PaLM 2-L+NExT is evalauted with 0-shot.

programming tasks that require reasoning about execution, such as code generation with partial execution contexts (Yin et al., 2023) or inferring program execution results (Nye et al., 2021), which we leave as important future work.

4.5 Experiments

Models. We evaluate NExT using PaLM 2-L (Unicorn) as the base LLM (Anil et al., 2023). Its finetuning API is publicly accessible on Google Cloud Vertex AI platform.

Datasets. We use two Python program repair benchmarks, **Mbpp-R** and **HumanEvalFix-Plus** (**HeFix+** hereafter). Mbpp-R is a new repair benchmark that we create from Mbpp (Austin et al., 2021b), a popular function-level Python code generation dataset. We create Mbpp-R by collecting LLM-generated incorrect code solutions to Mbpp problems, with a total of $10,047$ repair tasks for training and $1,468$ tasks (from a disjoint set of Mbpp problems) in the development for evaluation (§4.7.1). In addition to Mbpp-R, we also evaluate on HeFix+. HeFix+ is derived from HumanEvalFix (Muennighoff et al., 2023) which consists of 164 buggy programs for problems in the HumanEval dataset (Chen et al., 2021b). We further augment HumanEvalFix with the more rigorous test suites from EvalPlus (Liu et al., 2023) to obtain HeFix+. While both original datasets Mbpp and HumanEval feature function-level algorithmic code generation problems, problems from the two datasets may still differ in their topics, algorithms or data structures used. Therefore, we use HeFix+ to measure generalization ability without further finetuning.

Evaluating Code Fixes. We use pass@k (Chen et al., 2021b; Kulal et al., 2019), defined as the fraction of solved repair tasks using k samples ($k \leq 25$), to measure the end-to-end functional correctness of fixed programs with tests.

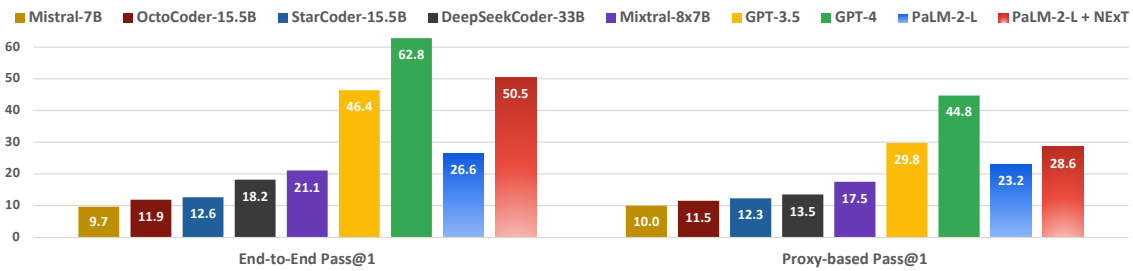


Figure 4.3: Greedy-decoding results on MBPP-R on PaLM 2-L+NExT and existing LLMs.

Evaluating Rationale Quality. Decoupling the quality of intermediate CoT rationales and downstream task performance (program repair PASS@k) is a non-trivial research question in LLM reasoning (Prasad et al., 2023), with most works on improving CoT reasoning still hill-climbing towards downstream task performance without evaluating intermediate rational quality (*e.g.,* Lightman et al. (2023)). To disentangle the evaluation of rationale quality from end-to-end repair accuracy, we propose an extrinsic **proxy-based evaluation** metric for rationales. Specifically, given a rationale r, we prompt a smaller LLM to solve the original repair task conditioning on r, and use the correctness of the predicted code fix (using greedy decoding) to approximate the quality of r. Intuitively, smaller LLMs would rely more on information from the rationale and could be more sensitive to its errors. Therefore, their performance could be a better indicator of rationale quality. We report averaged scores on two PaLM 2 variants for proxy-based evaluation: 1) a smaller general-purpose language model PaLM 2-S; and 2) PaLM 2-S* which is specialized in coding (Anil et al., 2023). Note that while we primarily use proxy-based metrics to evaluate rationales, we also perform human ratings of rationale quality (§4.5.2), with results in line with our proxy-based evaluation.

Hyperparameters. We perform temperature sampling ($T = 0.8$) with a sample size of 32 for training ($|S_j| = 32$ in Algo. 4) and PASS@k evaluation. In the first iteration in Algo. 4, we use PASS@1 estimated with these 32 samples as the filtering metric $\mathcal{M}(\cdot)$ to find challenging problems whose $\mathcal{M}(\cdot) \leq 10\%$ for training. We perform 10 iterations of NExT training and pick the best model using PASS@1 on the development set.

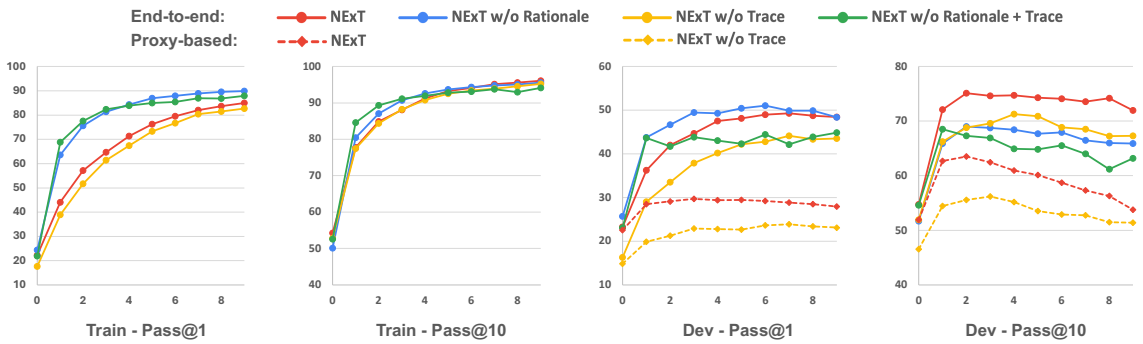


Figure 4.4: Ablations on removing rationales and/or traces during the iterative training of NExT. Note that different min/max values are taken for y-axis for clarify among different curves but consistent gridline intervals are used for easier comparison.

4.5.1 Main Results

In our experiments, we compare our model with strong LLMs (used in §4.3), analyze the impact of rationales and program traces, and perform generalization experiments on HeFix+ and human evaluation of rationale quality.

NExT improves program fix rate. We first compare the end-to-end program repair performance of PaLM 2-L before and after NExT training (**PaLM 2-L+NExT**) in Tab. 4.2 (*Left*). NExT leads to significant improvements on the end-to-end fix rates across the board, with a 26.1% absolute improvement on pass@1. Interestingly, the gain on pass@k is generally higher for smaller k. This might suggest that the model becomes more confident about program fixes after NExT training, while the sample diversity also improves, as indicated by improved pass@25. For reference, we also include results from GPT models. Notably, PaLM 2-L+NExT outperforms GPT-3.5 on all pass@k metrics.

NExT improves rationale quality. Table. 4.2 (*Right*) shows the improvements of PaLM 2-L+NExT on our proxy-based evaluation, where we approximate rationale quality using the performance of smaller LMs when conditioned on those rationales. Again, NExT yields consistent improvements across all pass@k metrics. This suggests that NExT improves PaLM 2-L's skill in reasoning with execution to solve Mbpp-R problems, leading to rationales that are more helpful for smaller LMs. In §4.9, we present a case study to demonstrate different reasoning strategies PaLM 2-L+NExT adopts to repair programs using execution information. As we later show in §4.5.2, our proxy-based metrics are also consistent with

human ratings, and rationales from PaLM 2-L+NExT are strongly preferred by annotators compared to those from PaLM 2-L.

PaLM 2-L+NExT outperforms strong LLMs. We compare PaLM 2-L+NExT with a series of strong LLMs from the preliminary study (§4.3) in Figure. 4.3. PaLM 2-L+NExT outperforms strong open-source LLMs by a minimum of 29.4% and 11.1% on end-to-end and proxy-based PASS@1 results, respectively, while on par with GPT-3.5. These results show that PaLM 2-L+NExT is a competitive model on program repair by reasoning with execution.

Learning to reason in natural language improves generalization and sample diversity. To further demonstrate the importance of using CoT reasoning in NExT self-training, we compare PaLM 2-L+NExT with a strong self-training-based program repair model implemented in NExT, which directly generates code fixes using runtime execution information without CoT reasoning. This ablation resembles SelfAPR (Ye et al., 2022), which also adopts self-training to iteratively synthesize data using unit test diagnostics, while our ablation uses traces with richer execution information. Fig. 4.4 shows model performance w.r.t. NExT training iterations. When trained without CoT reasoning (**NExT w/o rationale**), PaLM 2-L converges much faster on the training set, which is not surprising since the model only learns to generate code fixes without additional reasoning tasks such as explaining bugs in NL. However, on the DEV set, PaLM 2-L+NExT still outperforms this baseline in PASS@10 with comparable PASS@1 accuracy, and the gap on PASS@10 becomes larger with more iterations. This shows that by reasoning in natural language, PaLM 2-L+NExT generalizes much better to unseen MBPP-R problems with greater sample diversity. In Fig. 4.6 of §4.8, we also show that the gain from PaLM 2-L+NExT against this ablation on PASS@k is even more pronounced for larger $k > 10$, which suggests that learning to reason in CoT rationales improves sample diversity on program repair, similar to the findings on other code generation tasks (Yin et al., 2023).

Reasoning with execution traces is critical. To understand the importance of leveraging program traces to reason with execution, we compare with an ablation of NExT *without* using program traces, which follows the same procedure in Algo. 4 except that traces ϵ are not used to generate rationales in Line 5 (**NExT w/o traces**, Fig. 4.4). This variant

	Test w/ Trace		Test w/o Trace	
Methods	E2E	Proxy	E2E	Proxy
PaLM 2-L	23.2	22.5	19.0	14.8
+NExT (w/ trace)	$49.3_{+26.1}$	$28.8_{+6.3}$	$40.8_{+21.8}$	$19.5_{+4.7}$
+NExT w/o trace	–	–	$44.1_{+25.1}$	$23.9_{+9.1}$

Table 4.3: PaLM 2-L+NExT trained with traces outperforms PaLM 2-L when traces are absent at test time as shown in highlighted results. Results are on Mbpp-R; **Test w/ Trace:** results from Tab. 4.2.

can also be seen as a direct application of the rationale generation bootstrapping method in Zelikman et al. (2022), which trains a model on sampled rationales that lead to correct task solutions without relying on additional execution information. Without traces, PaLM 2-L is consistently worse than PaLM 2-L+NExT on the Dev set across iterations, both in terms of end-to-end fix rate and proxy-based metrics. This suggests that reasoning with execution information is critical for PaLM 2-L on program repair tasks. Interestingly, while the gap on the development set is significant, the two models achieve similar scores on the training set, which suggests that reasoning with pre-existing execution traces also help the model generalize better to unseen tasks at test-time.

Our model works without traces at test-time. While program traces are crucial for reasoning with execution, such execution information may not always be available at test time (*e.g.,* when execution is prohibitively expensive). To stress-test PaLM 2-L+NExT in scenarios where execution information is absent, we remove execution traces from its input at test time in Table. 4.3. PaLM 2-L+NExT still yields an end-to-end fix rate of 40.8%, which is an 21.8% improvement over the 3-shot PaLM 2-L baseline and is only 3.3% lower than NExT trained without traces, for which is tested in-distribution. The results from the proxy-based evaluation of rationales are also consistent with the fix rate.

Our model generalizes to HeFix+ at test-time. To further evaluate the generalization ability of PaLM 2-L+NExT, we test our model (trained on Mbpp-R) on HeFix+. Tab. 4.4 summarizes the results. NExT achieves reasonable generalization on HeFix+, outperforming the base PaLM 2-L model by a large margin (*i.e.,* 14.3% on end-to-end fix rate and 6.0% on proxy evaluation). Aligned with our previous findings on Mbpp-R in Fig. 4.4, reasoning with execution traces (*c.f.* w/o traces) improves fix rate and rationale

Models / pass@1	End-to-End	Proxy-based
Baselines w/ 3-shot prompting		
Mistral-7B*	12.8	16.5
OctoCoder-15.5B*	17.7	17.7
StarCoder-15.5B*	14.6	13.1
DeepSeekCoder-33B*	28.0	18.3
Mixtral-8x7B*	32.3	30.8
GPT-4	77.6	56.6
GPT-3.5	59.4	41.8
PaLM-2-L	32.2	31.9
PaLM-2-L w/o tracing†	30.3	30.4
PaLM 2-L+NExT	$42.5_{+10.3}$	$38.0_{+6.1}$
w/o tracing†	$38.1_{+7.8}$	$30.6_{+0.2}$
w/o rationale	$44.5_{+12.3}$	–
w/o tracing + rationale†	$31.4_{+1.1}$	–

Table 4.4: Generalization results on HeFix+. PaLM 2-L+NExT models are only trained with Mbpp-R. *obtained using greedy decoding; †no traces provided at test time.

	Explain bugs?			Suggest fixes?				
	✅	☑	❌	✅	☑	❌	Overall	Best?
GPT-3.5	43	26	35	44	16	44	51.9%	34.6%
PaLM 2-L	27	24	53	31	5	68	34.9%	6.7%
+NExT	48	24	32	42	6	56	50.5%	32.7%

Table 4.5: Results for human annotation of rationale quality. Base models use 3-shot prompting. Numbers under the questions are counts of ratings.

quality. Moreover, we remark that with iterative learning, PaLM 2-L+NExT is on par with the strong program repair method without CoT reasoning (w/o rationale), similar to the results on Mbpp-R. This is in contrast with our preliminary study in §4.3, where PaLM 2-L with CoT prompting is much worse than vanilla prompting without using rationales. Overall, these results indicate that PaLM 2-L+NExT could robustly generalize to out-of-distribution repair tasks without additional dataset-specific finetuning.

4.5.2 Human Evaluation of Rationale Quality

Our proxy-based evaluation suggests the extrinsic value of the CoT rationales from PaLM 2-L+NExT. We further conduct an intrinsic evaluation by manually rating the quality of model-predicted rationales on 104 sampled Mbpp-R repair tasks from the Dev set. Specifically, we ask raters to judge the quality of rationales generated by three models

(PaLM 2-L+NExT, PaLM 2-L and GPT-3.5) in a three-way side-by-side setting. Each rationale is rated in two aspects: (1) its helpfulness in explaining bugs (Q_1, *e.g.,* first two paragraphs in $\hat{r}$, Fig. 4.1), and (2) its helpfulness in suggesting code fixes (Q_2, *e.g.,* *"a fixed version that uses ..."* in $\hat{r}$). Each question has a three-scale answer (✅ Completely correct and very helpful; ☑ Partially correct with minor errors but still helpful; ❌ Incorrect and not helpful). We also compute an **overall score** of rationale quality using numeric values of $\{+1, 0.5, 0\}$ for the three scales and averaged over Q_1 and Q_2. Finally, we ask raters to pick a single **best choice** if there is not a clear tie. More details about our human evaluation pipeline is described in §4.7.3.

Tab. 4.5 summarizes the result. Compared to the base PaLM 2 model, PaLM 2-L+NExT generates significantly more high-quality rationales with correct explanations of bugs and fix suggestions. Additionally, compared to GPT-3.5, PaLM 2-L+NExT also has more rationales with correct bug explanations, while interestingly, GPT-3.5 generates more rationales with partially correct fix suggestions. We hypothesize that including more exemplars with detailed fix suggestions to our few-shot prompts during NExT training (§4.10) would help mitigate this issue. Nevertheless, the overall scores and rater-assigned best choice suggest that the rationales predicted by PaLM 2-L+NExT are of significantly higher quality compared to those from PaLM 2-L, and are on par with the predictions from GPT-3.5. Overall, this finding is in line with the proxy evaluation results in Fig. 4.3 (GPT 3.5 $\approx$ PaLM 2-L+NExT $\gg$ PaLM 2-L), suggesting that the latter is a reasonable surrogate metric for rationale quality. In §4.9, we present example generated rationales that show a variety of reasoning patterns.

4.6 Additional Details of NExT

4.6.1 Details for Inline Trace Representation

Definitions. A program $y \in \mathcal{Y}$ consists of a sequence of statements $\{u_1, ..., u_m\}$. And a program state h is a mapping between identifiers (*i.e.,* variable names) to values, *i.e.,* $h \in \{k \mapsto v | k \in \mathcal{K}, v \in \mathcal{V}\}$. Given an input to the program, an execution trace is defined as a sequence of program states, *i.e.,* $\epsilon = \{h_1, ..., h_t\}$, which are the results after executing

the statements with the *order of execution*, *i.e.,* $\{u_{e_1}, u_{e_2}, ..., u_{e_t}\}$. In this way, the relation between program statements and execution states can be seen as a function that maps from states to statements, *i.e.,* $h_i \mapsto u_{e_i}$, because each statement could be executed multiple times due to loops or recursion.

Program state representation. For typical programs, most of the variable values will stay the same between two adjacent states h_{i-1} and h_i. Thus to save tokens, we represent a state h_i only by the variables that have changed the value compared with the previous state h_{i-1}. And we use a reified variable state representation, *i.e.,* using the grammar for an init function in Python (*e.g.,* `lst=[1, 2, 3]`). Note that it is possible for a statement to have no effect on any traceable variables (*e.g.,* "`pass`", or "`print`", or "`lst[i]=lst[i]`"). To distinguish this case with unreached statements (*e.g.,* "`else`" branch that next got executed), we append a string "`NO_CHANGE`" instead. In addition to the variable state, we number all the states by the order of execution and prepend the ordinal number to the beginning of the state, *e.g.,* "`(1) odd_list=[]`" in Fig. 4.2.

Inline trace representation. To obtain the inline trace representation, we first group the program states in a trace ϵ by the corresponding program statements to collect a sequence of states for the same statement u_i as $H_i = \{h_j | u_{e_j} = u_i\}$, and we order the states in H_i by the execution order. For statements inside a loop body, or a function that is called recursively, the number of corresponding states can be very large. In order to further save tokens, if $|H_i| > 3$, we will only incorporate the first two states and the last state, and skip the ones in the middle. After that, we simply concatenate all the state representations with the semicolon "`;`" as the delimiter, and append it after the statement itself u_i following a hash "`#`" to note it as an inline comment. An example of the resulting representation is "`even_list.append(n) # (4) even_list=[1]; (8) even_list=[1, 3]; ...; (20) even_list=[1, 3, 5, 7, 9];`", as shown in Fig. 4.2.

Limitations. First of all, our tracing framework currently do not extend beyond native Python programs, thus it can not trace code that is not written in Python (*e.g.,* C code in `numpy`). One other limitation of our tracing representation is that for "`if`" conditions,

though it would be better to leave traces of "`(1) True; (2) True; (3); False;`", currently our tracing framework that based on the "`sys.settrace()`" hook of Python does not capture this. However, since we labeled all the states by the execution order, the LLMs can infer the conditions by the fact that certain branch is taken. Another limitation is the representation of Collections. Currently we still present all the elements in a collection, and empirically it works well with benchmarks as MBPP-R and HEFIX+. However, certain heuristics may be needed to skip certain elements (*e.g.,* like the one we use to skip certain states in a loop) to be more token efficient. For more complex objects (*e.g.,* Tensors, DataFrames), while we can define heuristics to represent key properties of those objects in traces (*e.g.,* "a float tensor of shape 128 x 64","a Dataframe with columns Name, Math, ..."), perhaps a more interesting idea would be to let the models decide which properties they would inspect and generate relevant code (*e.g.,* "`tensor.shape`" or "`df.head(3)`") to inspect them in a debugger or interpreter (*e.g.,* pdb). The same idea can be applied to longer programs, as the model can selectively decide which lines of code to inspect and create traces for, similar to how human developers debug programs. We will leave these as exciting future directions.

4.6.2 Details for Iterative Self-Training

Bootstrapping rationales and fixes via temperature sampling. To avoid potential "cold start" problem (Liang et al., 2018; Ni et al., 2020), for the first iteration, we use few-shot prompting with three exemplars (shown in §4.10) and set the sample size to 96. For all later iterations, we use zero-shot prompting as the model is already adapted to the style of the rationales and fixes after the first round of finetuning, and we set the sample size to 32. We set the sampling temperature $T = 0.8$ for all iterations.

Filtering rationales and fixes. Given the inputs in the prompt, we sample the rationale and fixes in tandem. To separate the natural language rationale and the program fix, we use an regular expression in Python to extract the content between two sets of three backticks (```), which is commonly used to note code blocks in markdown.[7] After we filter out the

7. For the strong LLMs that we used in this work, we did not observe any issue for following this style, which is specified in the few-shot prompt. The only exceptions are with GPT models, where they typically append the language (*i.e.,* "python") after the first set of backticks (*e.g.,* ```python), which we also handled

Methods	Use of Trace	Rationale Format	Model Fine-tuning
NExT	Input	Natural Language	Yes
Scratchpad (Nye et al., 2021)	Output	Scratchpad Repr.	Yes
Self-Debugging (Chen et al., 2023)	Output	Natural Language	No

Table 4.6: Comparison between the methods proposed in NExT, Scratchpad, and Self-Debugging.

Trace Repr.	Length Cutoff (# Tokens)							
	128	256	512	1,024	2,048	4,096	8,192	16,384
Inline (ours)	0.1%	7.3%	37.5%	78.9%	95.1%	98.5%	99.2%	99.5%
Scratchpad	0.0%	0.2%	15.1%	38.2%	60.1%	76.1%	85.1%	92.1%

Table 4.7: Percentage of MBPP-R examples that can be fit into different context windows using different trace representations (*i.e.,* ours and Nye et al. (2021)). Traces of all three tests are included.

rationales and fixes that are incorrect using the test cases, we create the training set by sub-sampling correct "(rationale, fix)" pairs to allow a maximum of 3 correct fixes with their rationales for each problem in MBPP-R. This is to balance the number of rationales and fixes for each problem and avoid examples from certain examples (typically easier ones) being overly represented in the training set.

4.6.3 Discussion with Previous Work

Here we discuss NExT in the context of two important previous work in the domain of reasoning about program execution, namely *Scratchpad* (Nye et al., 2021) and *Self-Debugging* (Chen et al., 2023). Such comparison is also characterized by Tab. 4.6.

***Scratchpad* and NExT.** Similarly to NExT, Nye et al. (2021) also proposed to use execution traces to help the LLMs to reason about program execution. However, Nye et al. (2021) aimed to generate these traces as intermediate reasoning steps at inference time, either via few-shot prompting or model fine-tuning. Yet in NExT, we use execution traces as part of the input to the LLMs, so they can directly use the execution states to ground the generated natural language rationales. Moreover, we choose to use natural language as the primary format for reasoning, which is more flexible and easier to be understood by the

with `regex`.

human programmers. We also perform a length comparison of our proposed inline trace representation with the scratchpad representation proposed in Tab. 4.7, and results show that our proposed inline trace representation is much more compact than scratchpad.

***Self-Debugging* and NExT.** Self-Debugging (Chen et al., 2023) is a seminal approach that also performs CoT reasoning over program execution to identify errors in code solutions. Different from NExT, Self-Debugging can optionally leverages high-level execution error messages to bootstrap CoT reasoning, while our method trains LLMs to reason with concrete step-wise execution traces. In addition, Self-Debugging also introduced a particular form of CoT rationales that resemble step-by-step traces in natural language. Notably, such rationales are generated by LLMs to aid the model in locating bugs by simulating execution in a step-by-step fashion. They are not the ground-truth execution traces generated by actually running the program. As we discussed in §4.11, in contrast, our model relies on existing traces from program execution. Since those traces already capture rich execution information, intuitively, the resulting CoT rationales in NExT could be more succinct and "to the point" without redundant reasoning steps to "trace" the program step-by-step by the model itself in order to recover useful execution information.

Finally, we remark that our "Test w/o Trace" setting in §4.5.1 shares similar spirits with the setup in Self-Debugging, as both methods perform CoT reasoning about execution without gold execution traces. From the results in Tab. 4.3, NExT also greatly improves the model's ability to repair programs even without using gold execution traces at test time. This may suggest that NExT can potentially improve the self-debugging skills of LLMs through iterative training, for which we leave as exciting future work to explore.

4.7 Experiment Setup Details

4.7.1 Creating Mbpp-R

The original Mbpp dataset (Austin et al., 2021b) consists of three splits, *i.e.,* train/dev/test sets of 374/90/500 Python programming problems. To increase the number of training example, we first perform a re-split of the original Mbpp dataset, by moving half of the test

data into the training split, resulting in 624/90/250 problems in the re-split dataset. Then for each MBPP problem in the re-split train and dev set, we collect a set of failed solutions from the released model outputs in Ni et al. (2023a). More specifically, we take the 100 samples for each problems, filter out those correct solutions, and keep the ones that do not pass all the tests. As different problems have various number of buggy solutions, we balance this out by keeping at most 20 buggy solutions for each MBPP problem.[8] This yields the MBPP-R dataset, with 10,047 repair tasks in the training set and 1,468 examples in the dev set.

4.7.2 Use of test cases.

For each program repair task, there is typically a set of open test cases that are used for debugging purposes, as well as a set of hidden test cases that are only used for evaluation of correctness. When we generate traces using test cases, we use only the open test cases and only feed the open test cases to the model as part of the prompt. Then when we evaluate the generated fix, we resort to all test cases (*i.e.,* open + hidden tests) and only regard a fix as correct when it passes all test cases. While the HUMANEVAL dataset makes this distinction between open and test cases, the MBPP dataset does not make such distinction. Thus for MBPP-R, we use all test cases both as the inputs and during evaluation. While this may lead to false positives when the fixes are overfit to the test cases, and we did find such case during human annotations.

4.7.3 Details of Human Annotation of Rationale Quality

We annotated model predictions on 104 sampled MBPP-R repair tasks from the DEV set. Those fix tasks are randomly sampled while ensuring that they cover all the 90 dev MBPP problems. All the tasks are pre-screened to be valid program repair problems. Annotation is performed in a three-way side-by-side setting. Models are anonymized and their order is randomized. Raters[9] are asked to judge the quality of rationales from three models

8. This actually biased the dataset towards harder problems as easier problems may not have more than 20 buggy solutions from 100 samples, thus it might be one of the reasons for repairing solutions in MBPP-R to be more challenging than generating code for the original MBPP dataset.

9. Four of the authors conducted the human evaluations.

For each task guid that you are going to work on:

1. Search for its entry in `annotation_data.html`. You will see a prompt and three model predictions (explanations and fixed code). In this task, **you will only rate the quality of the natural language explanations.** We also provided a correctness label of the fixed code. Please only use that for reference. Sometimes, the natural language explanation is correct even if the fixed code is wrong.
2. For each of the three models, please check its natural language explanation by answering the following questions. The three models we compare are: GPT3.5, NExT (ours) and PaLM-2-Large. Their order is randomized in each annotation task.

A typical explanation contains (1) explanation of why the code is wrong, and (2) how to fix the buggy code. you are going to rate the quality of these two parts separately. Here's an example explanation along with the code fix:

The issue with the provided code is that the string "element" is being inserted into the list instead of the value of the `element` variable. To fix this, you should use the `element` variable directly in the `insert` method. Here's the corrected code:

```
# Code fix, please ignore in this annotation
def add_str(tup, element):
    res = list(tup)
    for i in range(1, len(res)*2-1, 2):
        res.insert(i, element)
    return res
```

Questions:

Does the explanation correctly identify the error(s) in the original code?

1. Yes. It identifies all the errors in the code and there is no factual error in its explanation.
2. Partially correct. It only identifies some errors in the code, or the explanation has some factual errors. But overall it is still helpful.
3. No. The explanation has significant issues and is not helpful.

Does the explanation suggest a correct and helpful fix to the original code?

1. Yes. It gives a correct and helpful suggestion to fix the original code
2. Somewhat. It gives suggestions to fix some but not all the errors in the original code, or the suggestion has some errors but it's still helpful.
3. No, but Okay. It didn't suggest how to fix the code, but a developer should be able to easily correct the code given the explanation of the code error.
4. No. The suggestion is not correct at all, or there isn't any suggestion and it's not clear how to come up with a fix.

Figure 4.5: Instructions for the human annotators when annotating the quality of the model generated rationales.

(PaLM 2-L+NExT, PaLM 2-L and GPT-3.5) on the same MBPP-R problem. Each rationale is rated from two aspects: (1) its helpfulness in explaining bugs (Q_1 : *Does the rationale correctly explain bugs in the original code? e.g.,* first two paragraphs in $\hat{r}$, Fig. 4.1), and (2) its helpfulness in suggesting code fixes (Q_2 : *Does the rationale suggest a correct and helpful fix? e.g.,* *"a fixed version that uses ..."* in $\hat{r}$, Fig. 4.1).[10] Each question has a three-scale

10. We only rate the quality of rationales (not the fixed code), while we still show the predicted fixed code to raters for reference.

answer (✅ Completely correct and very helpful ; ☑ Partially correct with minor errors but still helpful; ❌ Incorrect and not helpful). In a pilot study, we find that fix suggestions could often be redundant if the rationale already contains detailed explanation of bugs such that a developer could easily correct the code without an explicit fix suggestion (*e.g.,* Example 2, §4.9). Therefore, for Q_2, we also consider such cases as correct (✅) if a model didn't suggest a fix in its rationale but the fix is obvious after bug explanations. We list our annotation guideline in Fig. 4.5. Note that for Q_2, both answers `(1)` and `(3)` are counted as correct (✅) answers.

4.8 Additional Experiment Results

Here we show the learning curve of NExT and all its ablations in Fig. 4.6. We also show the full results for Mbpp-R and HeFix+ in Tab. 4.8 and Tab. 4.9, respectively.

Learning CoT rationales further improves pass@25. From §4.5.1, we mention that learning to reason in natural language improves sample diversity, registering higher pass@10 than the baseline of finetuning for generating fixes only (**NExT w/o Rationale**). From Tab. 4.8 and Tab. 4.9, we can observe that such performance advantage is even larger with pass@25, with 7.6% improvements on Mbpp-R and 6.8% improvements on HeFix+.

Training on hard-only examples. One part of our data filtering pipeline is to only perform sampling and train on the samples from hard problems (§4.4). Here we discuss more about the benefits and potential issues of doing so, by presenting results on a "*w/o hard-only*" ablation, where the model learns from rationales and fixes from both hard and easy examples. Efficiency-wise, by only sampling on the hard example, which is around half of the problems, we greatly can accelerate the sampling process. And from results in Fig. 4.6, only training with hard example also comes with performance benefits under the iterative self-training framework. More specifically, we notice a non-trivial gap between the training curve of this "w/o hard-only" baseline and the rest of the ablations, especially for pass@10 and pass@25 performance on the training set. This means that the model trained on both easy and hard examples leads to more problems in the training set unsolved (*i.e.,*

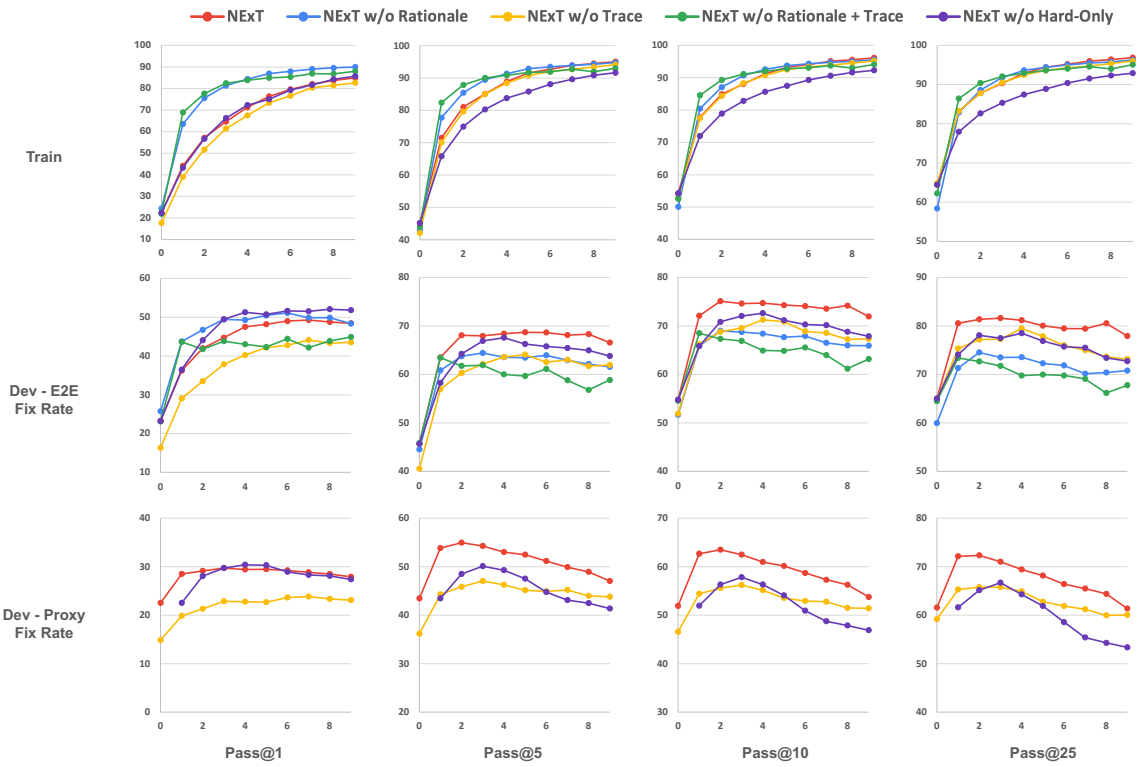


Figure 4.6: PASS@k performance on the train and dev sets of MBPP-R for NExT and all its ablations.

none of the samples are correct), and no learning signal can come from such problems. This also reflects on the dev set performance. While it is worth noticing that the end-to-end PASS@1 performance for "w/o hard-only" is slightly better than NExT trained only trained on hard examples, it performs worse in all other evaluations, with the trend of larger gaps with higher k values for PASS@k, especially for the proxy-based evaluation. This suggests that training on hard examples not only improves sample efficiency, but also improves the general fix rate as well as the quality of the generated rationales.

Proxy-based evaluation results are consistent with different proxy models. In the previous proxy-based evaluation §4.5.1, we report the proxy-based fix rates by averaging over the performance using PaLM 2-S and PaLM 2-S* as the proxy models. In Tab. 4.8 and Tab. 4.9, we show the separated results for different proxy models. From these results, we can observe that the relative rationale quality evaluated by different proxy models are largely consistent, with the stronger proxy model (PaLM 2-S*) having better proxy-based fix rates. In addition to the consistency we show with human annotations, this shows the

Models	End-to-End Fix Rate					Proxy-based Fix Rate (PaLM 2–S)					Proxy-based Fix Rate (PaLM 2–S*)				
	GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling			
	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25
GPT–3.5	46.4	42.9	65.0	70.7	76.7	27.9	24.7	46.1	54.5	64.6	31.8	28.5	51.5	59.5	68.2
GPT–3.5 w/o trace	47.1	46.8	65.9	70.7	75.7	27.2	25.6	47.0	55.5	64.7	30.9	30.2	53.0	60.7	68.8
GPT–4	62.8	63.2	75.1	78.5	82.7	41.8	42.2	64.5	71.0	76.6	47.8	47.4	68.5	73.9	79.0
GPT–4 w/o trace	51.3	44.8	68.5	73.4	78.5	29.4	27.1	54.2	63.4	72.2	34.9	32.0	60.3	68.5	75.7
PaLM 2-L	26.6	23.2	45.7	54.7	65.0	21.5	21.1	41.1	49.5	59.2	24.9	23.9	45.8	54.3	63.8
PaLM 2-L w/o trace	19.0	16.3	42.1	52.8	64.8	14.7	13.7	33.9	44.1	56.7	17.4	15.9	38.3	48.9	61.6
PaLM 2-L w/o rationale	27.5	25.7	44.5	51.7	60.0	–	–	–	–	–	–	–	–	–	–
PaLM 2-L w/o rationale + trace	23.8	23.1	45.8	54.6	64.5	–	–	–	–	–	–	–	–	–	–
NExT	50.5	49.3	68.1	73.5	79.4	25.3	26.1	46.8	54.4	62.9	31.8	31.6	53.0	60.2	68.1
test w/o trace	41.1	40.8	61.8	68.9	76.4	17.6	17.5	35.6	43.5	53.4	21.0	21.5	42.2	50.6	60.1
NExT w/o hard–only	52.9	52.1	65.0	68.8	73.4	23.5	25.1	38.6	44.0	50.9	30.0	29.7	44.1	49.7	55.9
test w/o trace	41.9	42.2	58.1	63.2	69.2	16.3	17.8	32.1	37.9	45.0	18.7	21.0	36.7	43.0	50.5
NExT w/o rationale	51.8	51.1	63.9	67.9	71.8	–	–	–	–	–	–	–	–	–	–
test w/o trace	43.7	43.0	57.2	61.7	66.3	–	–	–	–	–	–	–	–	–	–
NExT w/o trace	44.5	44.1	63.0	68.5	75.0	22.3	21.8	42.3	50.1	59.2	25.9	25.9	48.0	55.4	63.2
NExT w/o rationale w/o trace	46.3	44.9	58.9	63.2	67.8	–	–	–	–	–	–	–	–	–	–

Table 4.8: Full results on MBPP-R. "GD Acc." denotes PASS@1 evaluated with greedy decoding. All models in the top half are few-shot prompted while the bottom half shows the result of NExT and its ablations.

Models	End-to-End Fix Rate					Proxy-based Fix Rate (PaLM 2–S)					Proxy-based Fix Rate (PaLM 2–S*)				
	GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling				GD	PASS@k w/ Sampling			
	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25	Acc.	k=1	k=5	k=10	k=25
GPT-3.5	68.9	59.4	84.5	89.2	93.0	42.1	39.0	66.1	73.4	80.2	46.3	44.6	71.6	78.8	86.8
GPT-3.5 w/o trace	65.2	65.4	85.3	89.2	92.6	45.7	41.7	68.2	76.3	84.5	50.0	47.2	73.8	81.1	88.6
GPT-4	79.9	77.6	89.3	91.1	92.9	56.1	55.4	75.7	80.8	85.8	61.0	57.7	77.5	82.7	87.4
GPT-4 w/o trace	79.3	68.9	88.3	90.7	92.9	54.9	46.1	72.3	79.0	86.1	59.8	48.7	74.4	80.8	87.5
PaLM 2-L	43.3	32.2	64.3	73.8	81.5	32.9	28.9	59.0	69.2	79.1	43.3	34.9	65.8	74.3	82.9
PaLM 2-L w/o trace	38.4	30.3	61.9	72.9	83.3	25.6	27.8	56.2	66.0	76.6	31.1	33.0	63.5	72.7	81.8
PaLM 2-L w/o rationale	53.0	45.3	71.5	78.9	85.4	–	–	–	–	–	–	–	–	–	–
PaLM 2-L w/o rationale + trace	48.2	43.2	71.4	80.0	87.7	–	–	–	–	–	–	–	–	–	–
NExT	46.3	42.5	62.6	69.1	76.5	31.7	34.8	54.8	62.4	70.2	40.9	41.3	61.8	68.9	76.4
test w/o trace	42.7	41.2	62.9	70.6	79.5	26.8	26.4	48.0	56.1	64.2	36.0	32.6	55.7	64.4	72.8
NExT w/o hard-only	48.8	47.7	64.8	70.4	76.6	32.9	37.2	50.8	55.5	61.9	41.5	42.4	56.3	60.8	66.9
test w/o trace	47.6	44.2	64.4	70.4	75.5	31.7	33.3	46.9	51.4	57.3	38.4	38.5	54.6	59.2	63.9
NExT w/o rationale	47.6	44.5	58.9	63.7	69.4	–	–	–	–	–	–	–	–	–	–
test w/o trace	46.3	44.7	60.4	65.2	70.2	–	–	–	–	–	–	–	–	–	–
NExT w/o trace	40.9	38.1	59.1	65.3	71.5	29.3	26.9	52.1	61.1	71.5	33.5	34.4	63.1	70.8	77.4
NExT w/o rationale w/o trace	30.5	31.4	44.6	49.0	54.1	–	–	–	–	–	–	–	–	–	–

Table 4.9: Full results on HEFIX+. Same notations from Tab. 4.8 apply.

robustness of our proposed proxy-based evaluation method for measuring CoT rationale quality.

4.9 Case Study

In this section we present a set of examples to showcase how PaLM 2-L+NExT reasons with program execution to solve MBPP-R problems. We discover several reasoning patterns the model exhibits that leverage trace information to identify and explain bugs in programs. First, as shown in **Example 1**, the model could refer to exceptions or error messages in the trace (eg in `Trace 2`) to explain bugs in the code. Next, **Example 2** shows that the model could also leverage variable states in the trace (*e.g.,* in `Trace 2`) and compare them with the expected values to locate the cause of bugs. Besides, the `NO_CHANGE` annotations for variables whose values are preserved after execution of a step could also help the model

explain the execution process in the rationale (*e.g.,* `(3)NO_CHANGE` ↦ *"the first sublist is already sorted"*). Perhaps a more interesting scenario is when the model reasons over multiple steps of computation to track down the cause of a bug. In **Example 3**, the model attempts to trace the computation of steps 2 - 4 in `Trace 1` to explain why the sum is a float instead of an integer. Another example is **Example 4**, where the model summarizes the loop iterations in steps 2 - 9 of `Trace 1` to explain the cause of the missing last element in the result list. Interestingly, while the model is able to reason over multiple steps of execution in its rationales, as the reasoning chain becomes longer, it is more likely to contain minor logical errors, as highlighted in the rationales of Examples 3 and 4.

Besides minor logical errors in CoT reasoning, we also identify several more pronounced error modes. First, even if the NL rationale looks valid, the actual code fix might be incorrect (**Example 4**). Second, for cases where there are multiple bugs in the program, the model might only focus on the more obvious bugs during reasoning (*e.g.,* explaining the reason of `IndexError` in `Trace 2` of **Example 5**), while ignoring the other more nuanced bugs that are buried in the trace (*e.g.,* Incorrect outputs in `Trace 1` and `Trace 3` caused by zero-initialized `freq_arr`). This could be especially problematic when there are hidden bugs *not* captured by the trace, which could happen when a buggy program crashes in the middle of execution and the rest the of the code is not covered in the trace. Since the model could be more reliant on trace information after fine-tuning, it might be more likely to miss such implicit bugs. As an example, in **Example 6** the model only noticed the `TypeError` in the trace, without spotting that the code that computes the end index in the output tuple is off by one.

Example 1: An example where our model explains exceptions in the trace

---

Task: Write a function to remove leading zeroes from an ip address

Buggy Code:

```
def removezero_ip(ip):
    ip_list = ip.split(".")
    for i in range(4):
        ip_list[i] = str(int(ip_list[i]))
    return ".".join(ip_list)
```

Test Cases:

```
assert removezero_ip("216.08.094.196")==('216.8.94.196')
assert removezero_ip("12.01.024")==('12.1.24')
assert removezero_ip("216.08.094.0196")==('216.8.94.196')
```

Traces:

```
# Trace 1:
def removezero_ip(ip):  # (0) ip='216.08.094.196';
    ip_list = ip.split(".")  # (1) ip_list=['216', '08', '094', '196'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; ...; (10) NO_CHANGE;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['216', '8', '094', '196'];
(7) ip_list=['216', '8', '94', '196']; (9) NO_CHANGE;
    return ".".join(ip_list)  # (11) __return__='216.8.94.196';

assert removezero_ip("216.08.094.196")==('216.8.94.196')

# Trace 2:
def removezero_ip(ip):  # (0) ip='12.01.024';
    ip_list = ip.split(".")  # (1) ip_list=['12', '01', '024'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; (8) i=3;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['12', '1', '024']; (7)
ip_list=['12', '1', '24']; (9)  __exception__ = IndexError('list index out of range')
    return ".".join(ip_list)

assert removezero_ip("12.01.024")==('12.1.24')

# Trace 3:
def removezero_ip(ip):  # (0) ip='216.08.094.0196';
    ip_list = ip.split(".")  # (1) ip_list=['216', '08', '094', '0196'];
    for i in range(4):  # (2) i=0; (4) i=1; (6) i=2; ...; (10) NO_CHANGE;
        ip_list[i] = str(int(ip_list[i]))  # (3) NO_CHANGE; (5) ip_list=['216', '8', '094', '0196'];
(7) ip_list=['216', '8', '94', '0196']; (9) ip_list=['216', '8', '94', '196'];
    return ".".join(ip_list)  # (11) __return__='216.8.94.196';

assert removezero_ip("216.08.094.0196")==('216.8.94.196')
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to write a function to remove leading zeroes from an IP address.

Now let's look at the program. The program first splits the IP address into a list of strings, and then iterates through the list to convert each string to an integer, and then back to a string. This will remove leading zeroes.

However, the problem is that it iterates through the list for 4 times, but the length of the list might not be 4. For example, when `ip="12.01.024"`, the length of the list is only 3. Thus when `i=3`, it will throw an `IndexError` .

Here is a simple fix:

```
def removezero_ip(ip):
    ip_list = ip.split(".")
    for i in range(len(ip_list)):
        ip_list[i] = str(int(ip_list[i])).lstrip('0')
    return ".".join(ip_list)
```

Example 2: An example where our model uses variable states to solve the task

---

Task: Write a function to sort each sublist of strings in a given list of lists

Buggy Code:

```
def sort_sublists(words):
    words.sort(key = len)
    for w in words:
        w.sort()
    return words
```

Test Cases:

```
assert sort_sublists([['green', 'orange'], ['black', 'white'], ['white', 'black', 'orange']])==[['green', 'orange'], ['black', 'white'], ['black', 'orange', 'white']]
assert sort_sublists([['green', 'orange'], ['black'], ['green', 'orange'], ['white']])==[['green', 'orange'], ['black'], ['green', 'orange'], ['white']]
assert sort_sublists([['a','b'],['d','c'],['g','h'] , ['f','e']])==[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']]
```

Traces:

```
# Trace 1:
def sort_sublists(words):  # (0) words=[['green', 'orange'], ['black', 'white'], ['white', 'black', '
orange']];
    words.sort(key = len)  # (1) NO_CHANGE;
    for w in words:  # (2) w=['green', 'orange']; (4) w=['black', 'white']; (6) w=['white', 'black', '
orange']; (8) NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) NO_CHANGE; (7) w=['black', 'orange', 'white'], words=[['green',
 'orange'], ['black', 'white'], ['black', 'orange', 'white']];
    return words  # (9) __return__=[['green', 'orange'], ['black', 'white'], ['black', 'orange', '
white']];

assert sort_sublists([['green', 'orange'], ['black', 'white'], ['white', 'black', 'orange']])==[['
green', 'orange'], ['black', 'white'], ['black', 'orange', 'white']]

# Trace 2:
def sort_sublists(words):  # (0) words=[['green', 'orange'], ['black'], ['green', 'orange'], ['white
']];
    words.sort(key = len)
# (1) words=[['black'], ['white'], ['green', 'orange'], ['green', 'orange']];
    for w in words:  # (2) w=['black']; (4) w=['white']; (6) w=['green', 'orange']; ...; (10)
NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) NO_CHANGE; (7) NO_CHANGE; (9) NO_CHANGE;
    return words  # (11) __return__=[['black'], ['white'], ['green', 'orange'], ['green', 'orange']];

assert sort_sublists([['green', 'orange'], ['black'], ['green', 'orange'], ['white']])==[['green', '
orange'], ['black'], ['green', 'orange'], ['white']]  # (12)  __exception__ = AssertionError()

# Trace 3:
def sort_sublists(words):  # (0) words=[['a', 'b'], ['d', 'c'], ['g', 'h'], ['f', 'e']];
    words.sort(key = len)  # (1) NO_CHANGE;
    for w in words:  # (2) w=['a', 'b']; (4) w=['d', 'c']; (6) w=['g', 'h']; ...; (10) NO_CHANGE;
        w.sort()  # (3) NO_CHANGE; (5) w=['c', 'd'], words=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['f',
'e']]; (7) NO_CHANGE; (9) w=['e', 'f'], words=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']];
    return words  # (11) __return__=[['a', 'b'], ['c', 'd'], ['g', 'h'], ['e', 'f']];

assert sort_sublists([['a','b'],['d','c'],['g','h'] , ['f','e']])==[['a', 'b'], ['c', 'd'], ['g', 'h'
], ['e', 'f']]
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to sort each sublist of strings in a given list of lists.

Now let's look at the program. The program first sorts the list of lists by the length of each sublist , and then sorts each sublist. The first assertion passes as the first sublist is already sorted, and the second sublist is also sorted, and the third sublist is sorted in the for loop.
The second assertion fails as the list of lists is sorted by the length of each sublist, and thus the return value is not the same as the input. The third assertion passes as the first and the last sublists are already sorted, and the second and the third sublists are sorted in the for loop.

The problem is that the program sorts the list of lists by the length of each sublist, which is not required by the problem spec. Thus we can simply remove the first line of the function.

Here is the corrected code:

```
def sort_sublists(words):
    for w in words:
        w.sort()
    return words
```

Example 3: An example where PaLM 2-L+NExT reasons over multiple steps of execution

---

Task: Write a function to get the sum of a non-negative integer

Buggy Code:

```
def sum_digits(number):
    sum = 0
    while(number):
        sum += number number = number / 10
    return sum
```

Test Cases:

```
assert sum_digits(345)==12
assert sum_digits(12)==3
assert sum_digits(97)==16
```

Traces:

```
# Trace 1:
def sum_digits(number):  # (0) number=345;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number number = number / 10  # (3) number=34.5; (5) number=3.45; (7) number
=0.34500000000000003; ...; (655) number=0.0;
    return sum  # (656) __return__=13.333333333333333;

assert sum_digits(345)==12  # (657)  __exception__ = AssertionError()

# Trace 2:
def sum_digits(number):  # (0) number=12;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number number = number / 10  # (3) number=1.2; (5) number=0.12; (7) number=0.012; ...;
(651) number=0.0;
    return sum  # (652) __return__=3.3333333333333335;

assert sum_digits(12)==3  # (653)  __exception__ = AssertionError()

# Trace 3:
def sum_digits(number):  # (0) number=97;
    sum = 0  # (1) sum=0;
    while(number):
        sum += number number = number / 10  # (3) number=9.7; (5) number=0.97; (7) number=0.097; ...;
(653) number=0.0;
    return sum  # (654) __return__=17.77777777777777;

assert sum_digits(97)==16  # (655)  __exception__ = AssertionError()
```

Rationale and Code Solution: (code fix is correct ✅)

The problem spec asks to write a function to get the sum of all digits in a non-negative integer.

Now let's look at the program. The program seems to be correct at the first glance. It first initializes the sum to 0, and then it uses a while loop to get each digit in the number and add it to the sum. However, if we look at the output of the program, we will find that the sum is not an integer .

This is because the program does not convert the number to an integer before it gets the digits. For example, for the first assertion, when ‘number=345‘, the first digit ‘5‘ is added to the sum and ‘sum=5‘. Then ‘number‘ is updated to ‘34.5‘, and the second digit ‘4‘ is added to the sum and ‘sum=9.5‘ And so on .

Here is a fixed version:

```
def sum_digits(number):
    number = int(number)
    sum = 0
    while(number):
        sum += number number = number // 10
    return sum
```

Example 4: An example where PaLM 2-L+NExT reasons over multiple steps of execution

---

Task: Write a function to convert tuple into list by adding the given string after every element.

Buggy Code:

```
def add_str(test_tuple, add_str):
    res = list(test_tuple)
    for i in range(1,len(test_tuple)):
        res.insert(i*2-1,add_str)
    return res
```

Test Cases:

```
assert add_str((5, 6, 7, 4, 9) , "FDF") == [5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9, 'FDF']
assert add_str((7, 8, 9, 10) , "PF") == [7, 'PF', 8, 'PF', 9, 'PF', 10, 'PF']
assert add_str((11, 14, 12, 1, 4) , "JH") == [11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4, 'JH']
```

Traces:

```
# Trace 1:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[5, 6, 7, 4, 9];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; ...; (10) NO_CHANGE;
        res.insert(i*2-1,add_str)  #
(3) res=[5, 'FDF', 6, 7, 4, 9]; (5) res=[5, 'FDF', 6, 'FDF', 7, 4, 9];

(7) res=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 9]; (9) res=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9];
    return res  # (11) __return__=[5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9];

assert add_str((5, 6, 7, 4, 9) , "FDF") == [5, 'FDF', 6, 'FDF', 7, 'FDF', 4, 'FDF', 9, 'FDF']  # (12)
 __exception__ = AssertionError()

# Trace 2:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[7, 8, 9, 10];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; (8) NO_CHANGE;
        res.insert(i*2-1,add_str)  # (3) res=[7, 'PF', 8, 9, 10]; (5) res=[7, 'PF', 8, 'PF', 9, 10];
(7) res=[7, 'PF', 8, 'PF', 9, 'PF', 10];
    return res  # (9) __return__=[7, 'PF', 8, 'PF', 9, 'PF', 10];

assert add_str((7, 8, 9, 10) , "PF") == [7, 'PF', 8, 'PF', 9, 'PF', 10, 'PF']  # (10)  __exception__ =
 AssertionError()

# Trace 3:
def add_str(test_tuple, add_str):
    # Write your code here
    res = list(test_tuple)  # (1) res=[11, 14, 12, 1, 4];
    for i in range(1,len(test_tuple)):  # (2) i=1; (4) i=2; (6) i=3; ...; (10) NO_CHANGE;
        res.insert(i*2-1,add_str)  # (3) res=[11, 'JH', 14, 12, 1, 4]; (5) res=[11, 'JH', 14, 'JH',
12, 1, 4]; (7) res=[11, 'JH', 14, 'JH', 12, 'JH', 1, 4]; (9) res=[11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH
', 4];
    return res  # (11) __return__=[11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4];

assert add_str((11, 14, 12, 1, 4) , "JH") == [11, 'JH', 14, 'JH', 12, 'JH', 1, 'JH', 4, 'JH']  # (12)
 __exception__ = AssertionError()
```

Rationale and Code Solution: (code fix is wrong ✗)

The problem spec asks to convert a tuple to a list and add a given string after every element.

Now let's look at the program. The program first converts the tuple to a list, which is good. Then it iterates from 1 to the length of the list, and inserts the given string at index `i*2-1`. This seems correct at the first glance, but

the problem is that it will not add the string after the last element. For example, when ‘test_tuple=(5, 6, 7, 4, 9)‘, the length of the list is 5, and the for loop will iterate from 1 to 4. Thus the string will be inserted at index 1, 3, 5, and 7, which corresponds to after the second, fourth and sixth element.

The string will not be inserted after the last element (i.e., the ninth element).

Here is a simple fix:

```
def add_str(test_tuple, add_str):
    res = list(test_tuple)
    for i in range(len(test_tuple)):
        res.insert(i*2,add_str)
    return res
```

Example 5: An example where PaLM 2-L+NExT failed to identify all the bugs in the program

---

Task: Write a python function to find the difference between highest and least frequencies in a given array.

Buggy Code:

```
def find_Diff(nums, n):
    freq_arr = [0 for i in range(n)]
    for i in range(len(nums)):
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1
    return max(freq_arr) - min(freq_arr)
```

Test Cases:

```
assert find_Diff([1,1,2,2,7,8,4,5,1,4],10) == 2
assert find_Diff([1,7,9,2,3,3,1,3,3],9) == 3
assert find_Diff([1,2,1,2],4) == 0
```

Traces:

```
# Trace 1:
def find_Diff(nums, n):  # (0) n=10, nums=[1, 1, 2, 2, 7, 8, 4, 5, 1, 4]; (1) .0=<range_iterator
object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (13) freq_arr=[0, 0, 0, 0, 0,
0, 0, 0, 0, 0];
    for i in range(len(nums)):  # (14) i=0; (16) i=1; (18) i=2; ...; (34) NO_CHANGE;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (15) freq_arr=[0, 1, 0, 0, 0, 0, 0, 0, 0, 0];
(17) freq_arr=[0, 2, 0, 0, 0, 0, 0, 0, 0, 0]; (19) freq_arr=[0, 2, 1, 0, 0, 0, 0, 0, 0, 0]; ...; (33)
freq_arr=[0, 3, 2, 0, 2, 1, 0, 1, 1, 0];
    return max(freq_arr) - min(freq_arr)  # (35) __return__=3;

assert find_Diff([1,1,2,2,7,8,4,5,1,4],10) == 2  # (36) __exception__ = AssertionError()

# Trace 2:
def find_Diff(nums, n):  # (0) n=9, nums=[1, 7, 9, 2, 3, 3, 1, 3, 3]; (1) .0=<range_iterator object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (12) freq_arr=[0, 0, 0, 0, 0,
0, 0, 0, 0];
    for i in range(len(nums)):  # (13) i=0; (15) i=1; (17) i=2;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (14) freq_arr=[0, 1, 0, 0, 0, 0, 0, 0, 0]; (16)
freq_arr=[0, 1, 0, 0, 0, 0, 0, 1, 0]; (18)  __exception__ = IndexError('list index out of range')
    return max(freq_arr) - min(freq_arr)

assert find_Diff([1,7,9,2,3,3,1,3,3],9) == 3

# Trace 3:
def find_Diff(nums, n):  # (0) n=4, nums=[1, 2, 1, 2]; (1) .0=<range_iterator object>;
    freq_arr = [0 for i in range(n)]  # (2) i=0; (3) i=1; (4) i=2; ...; (7) freq_arr=[0, 0, 0, 0];
    for i in range(len(nums)):  # (8) i=0; (10) i=1; (12) i=2; ...; (16) NO_CHANGE;
        freq_arr[nums[i]] = freq_arr[nums[i]] + 1  # (9) freq_arr=[0, 1, 0, 0]; (11) freq_arr=[0, 1,
1, 0]; (13) freq_arr=[0, 2, 1, 0]; (15) freq_arr=[0, 2, 2, 0];
    return max(freq_arr) - min(freq_arr)  # (17) __return__=2;

assert find_Diff([1,2,1,2],4) == 0  # (18) __exception__ = AssertionError()
```

Rationale and Code Solution: (code fix is wrong ✗)

The problem spec asks to find the difference between the highest and the least frequencies in a given array.

Now let's look at the program. The program first creates a frequency array of length `n`, and then iterates through the input array to count the frequency of each number. It then returns the difference between the maximum and the minimum frequencies in the frequency array.

The program looks good at the first glance.
However, the problem is that the length of the frequency array is ‘n‘, which is the length of the input array. However, the numbers in the input array can be larger than ‘n‘, and thus the frequency array will not be able to accommodate all the numbers in the input array. For example, in the second assertion, the input array is `[1, 7, 9, 2, 3, 3, 1, 3, 3]`, and the length of the frequency array is also `9`. When `nums[i]=9`, it will try to access `freq_arr[9]`, which is out of bound, and thus an `IndexError` is thrown.

Here is a simple fix:

```
def find_Diff(nums, n):
    freq_arr = {}
    for i in range(len(nums)):
        if nums[i] in freq_arr:
            freq_arr[nums[i]] += 1
        else:
            freq_arr[nums[i]] = 1
    max_freq = 0
    min_freq = 0
    for num, freq in freq_arr.items():
        if freq > max_freq:
            max_freq = freq
        if freq < min_freq or min_freq == 0:
            min_freq = freq
    return max_freq - min_freq
```

Example 6: An example where PaLM 2-L+NExT failed to identify a bug not revealed in the trace

---

Task: Write a function to search a literals string in a string and also find the location within the original string where the pattern occurs.

Buggy Code:

```
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)

    return match
```

Test Cases:

```
assert search_literal('python','python programming language')==(0,6)
assert search_literal('programming','python programming language')==(7,18)
assert search_literal('language','python programming language')==(19,27)
```

Traces:

```
# Trace 1:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ =
TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('python','python programming language')==(0,6)

# Trace 2:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ =
TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('programming','python programming language')==(7,18)

# Trace 3:
def search_literal(pattern, text):
    match = tuple(text.find(pattern), text.find(pattern) + len(pattern) - 1)  # (1)  __exception__ =
TypeError('tuple expected at most 1 argument, got 2')

    return match

assert search_literal('language','python programming language')==(19,27)
```

Rationale and Code Solution: (code fix is wrong ✗)

Now let's look at the program. The program looks good at the first glance as `text.find(pattern)` is the correct way to find the location of a literal string in a given string. But the problem is that it does not use the `tuple()` function correctly. The `tuple()` function takes a single iterable as input, and returns a tuple with the elements from the iterable.
But in the program, it takes two arguments, which will throw a ‘TypeError‘ of ‘tuple expected at most 1 argument, got 2‘.

Here is a simple fix:

```
def search_literal(pattern, text):
    match = tuple(text.find(pattern),)
    match = match + (text.find(pattern) + len(pattern) - 1,)

    return match
```

4.10 Full Prompt

Here we show the instruction and all three few-shot examples used in this work. The rationales and fixes in the few-shot examples are hand annotated by the authors. Each exemplar has three unit tests and their traces.

````
We are doing pair programming, I will give you the problem definition and my implementation, which does not pass all test cases. And you need to help me to fix my code.

# Example 1

Here is the specification in natural language and the program that I wrote:
```
# Write a function to find the similar elements from the given two tuple lists.
def similar_elements(t1, t2):
        s1 = set(t1)
        s2 = set(t2)
        return s1.intersection(s2)

```
Here are the assertions I need to pass:
```
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)
assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)
````
````

```
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

```
#################### cheatsheet starts ####################
def similar_elements(t1, t2):   # (0) t1=(3, 4, 5, 6), t2=(5, 7, 4, 10);
        s1 = set(t1)   # (1) s1={3, 4, 5, 6};
        s2 = set(t2)   # (2) s2={10, 4, 5, 7};
        return s1.intersection(s2)  # (3) __return__={4, 5}


assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)  #
--------------------
def similar_elements(t1, t2):   # (0) t1=(1, 2, 3, 4), t2=(5, 4, 3, 7);
        s1 = set(t1)   # (1) s1={1, 2, 3, 4};
        s2 = set(t2)   # (2) s2={3, 4, 5, 7};
        return s1.intersection(s2)  # (3) __return__={3, 4}


assert similar_elements((1, 2, 3, 4),(5, 4, 3, 7)) == (3, 4)  #
--------------------
def similar_elements(t1, t2):   # (0) t1=(11, 12, 14, 13), t2=(17, 15, 14, 13);
        s1 = set(t1)   # (1) s1={11, 12, 13, 14};
        s2 = set(t2)   # (2) s2={17, 13, 14, 15};
        return s1.intersection(s2)  # (3) __return__={13, 14}


assert similar_elements((11, 12, 14, 13),(17, 15, 14, 13)) == (13, 14)  #
#################### cheatsheet ends ####################
```

Given the above information, can you tell me why my code is wrong, and also provide a fixed version?

The problem spec asks to find the similar elements in the two tuples, from the assertion we can see that it essentially wants the common elements from the two tuples. It also seems that the common elements should be returned in a tuple with ascending order.

````
Now let's look at the program. While it is correct and efficient to use `intersection()` to find the common elements between two sets, it will also return a set. Take the first assertion as an example, `s1` will first be converted to a set like `{3, 4, 5, 6}` and similarly for `s2`, then the return value will be `{4, 5}`. Though the elements are right, the expected return value is a tuple and not a set. Thus we only need to further convert the sets to tuples using `tuple()`.

Here is the corrected code:
```
def similar_elements(t1, t2):
        s1 = set(t1)
        s2 = set(t2)
        return tuple(s1.intersection(s2))
```

# Example 2

Here is the specification in natural language and the program that I wrote:
```
# Write a python function to identify non-prime numbers.
def is_not_prime(n):
  if n==1:
    return False
  else:
    for i in range(2,n//2+1):
      if nreturn False
      else:
        return True
```
Here are the assertions I need to pass:
```
assert is_not_prime(2) == False
assert is_not_prime(10) == True
assert is_not_prime(35) == True
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

#################### cheatsheet starts ####################
def is_not_prime(n):  # (0) n=2;
  if n==1:  #
````

```
    return False
  else:
    for i in range(2,n//2+1):  # (1) __return__=None
      if nreturn False
      else:
        return True


assert is_not_prime(2) == False  #
--------------------
def is_not_prime(n):  # (0) n=10;
  if n==1:  #
    return False
  else:
    for i in range(2,n//2+1):  # (1) i=2;
      if nreturn False  # (2) __return__=False
      else:
        return True


assert is_not_prime(10) == True  #
--------------------
def is_not_prime(n):  # (0) n=35;
  if n==1:  #
    return False
  else:
    for i in range(2,n//2+1):  # (1) i=2;
      if nreturn False
      else:
        return True  # (2) __return__=True


assert is_not_prime(35) == True  #
#################### cheatsheet ends ####################

Given the above information, can you tell me why my code is wrong, and also provide a fixed
↪  version?

The problem spec asks to write a Python function that identifies non-prime numbers. Non-prime
↪  numbers are numbers that are divisible by numbers other than 1 and itself. Note that 0 and 1
↪  are also non-prime numbers.

```

````
Now let's check the program. Let's run the program through some examples. For the first assertion, when `n=2`, it will enter the `else` branch, and then in the for loop, since the condition is `i in range(2, n//2+1)`, which is `range(2, 2)`, it will actually not enter the loop since the condition is not met in the first place. And then it will reach the end of the function, and the return value will be `None`. And for the third assertion, when `n=35`, it also enters the `else` branch, and it then enters the loop with `i=2` as `n//2+1=18 > 2`. Then in the next step, since `n

While it is not obvious how to correct this program, another way to implement the `is_not_prime()` function is just to first create a helper function to identify prime numbers as `is_prime()`, and then simply `return not is_prime(x)`.

Here is a re-implementation:
```
def is_prime(x):
  if x <= 1:
    return False
  else:
    for i in range(2,x):
      if xreturn False
    return True

def is_not_prime(x):
  return not is_prime(x)
```

# Example 3

Here is the specification in natural language and the program that I wrote:
```
# Write a function to find the largest integers from a given list of numbers using heap queue algorithm.
def heap_queue_largest(list1,n):
  largest=heapq.nlargest(n,list1)
  return largest

```
Here are the assertions I need to pass:
```
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],2)==[85, 75]
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],5)==[85, 75, 65, 58, 35]
````
````

```
```

Here is a *cheatsheet* for the gold execution trace for each assertions, use the information here but pretend you did all the reasoning about execution yourself.

```
#################### cheatsheet starts ####################
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=3;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]
--------------------
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=2;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],2)==[85, 75]
--------------------
def heap_queue_largest(list1,n):  # (0) list1=[25, 35, 22, 85, 14, 65, 75, 22, 58], n=5;
  largest=heapq.nlargest(n,list1)  # (1)  __exception__ = NameError("name 'heapq' is not defined")
  return largest


assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],5)==[85, 75, 65, 58, 35]
#################### cheatsheet ends ####################
```

Given the above information, can you tell me why my code is wrong, and also provide a fixed version?

The problem spec asks to find `n` largest integers in a given list. It also suggests that the heap queue shall be used.

Now let's look at the program. The program looks good at the first glance as `heapq.nlargest()` is the correct way to get the `n` largest integers from an iterable (e.g., a list). But the problem is that it does not import the `heapq` package first. Thus all three assertions will fail at the first line of the function, and throw a `NameError` of `name 'heapq' is not defined`.

````
193
194 Here is a simple fix:
195 ```
196 import heapq
197 def heap_queue_largest(list1,n):
198   largest=heapq.nlargest(n,list1)
199   return largest
200 ```
````

4.11 Related Work

Reasoning about Program Execution Several lines of research has explored learning methods to reason about program execution. Program synthesis systems often leverage the execution states of partially generated programs (Chen et al., 2021e; Shi et al., 2022b; Shin et al., 2018; Wang et al., 2018a) or the next execution subgoals (Shi et al., 2024) to guide search in sequence-to-sequence models. There has also been work on training neural networks to mimic program execution, like a learned interpreter (Bieber et al., 2020; Nye et al., 2021; Zaremba and Sutskever, 2014), often with specialized neural architectures to model the data flow of program execution (Bieber et al., 2022; Bosnjak et al., 2016; Gaunt et al., 2016; Graves et al., 2014). Instead of using domain-specific architectures to encode and reason about program execution, our work focuses on teaching LLMs to reason with execution in natural language. In particular, *Scratchpad* (Nye et al., 2021) and *Self-Debugging* (Chen et al., 2023) are two notable works that also models execution traces using LLMs. The core difference is that these methods focus on predicting reasoning chains that contain trace information, such as executed lines with variable states (Nye et al., 2021) or their natural language summaries (Chen et al., 2023). On the other hand, NExT aims to leverage existing execution traces from a runtime to aid the reasoning process, which often leads to more compact rationales tailored for downstream tasks. We present a more detailed comparison and discussion on NExT and these related works in §4.6.3.

Program Repair Several works in program repair have leveraged execution information such as traces (Bouzenia et al., 2023; Gupta et al., 2020) or test diagnostics (Xia and Zhang, 2023; Ye et al., 2022). Different from Bouzenia et al. (2023) which represents traces by

directly pairing unrolled executed lines with their variable states, NExT inlines indexed variable states as code comments, which is more token efficient while preserving the original code structure. Similar to NExT, Ye et al. (2022) construct synthetic self-training data using test execution results, while our approach generates both NL rationales and fixed programs with better interpretability. Recently, LLMs have been applied to program repair (Fan et al., 2022; Jiang et al., 2023b; Paul et al., 2023; Sobania et al., 2023; Xia and Zhang, 2022; Xia et al., 2023) and they are shown to be more effective that traditional or other learning based Automated Program Repair (APR) methods (Xia et al., 2023). Among them, Kang et al. (2023) uses a ReAct-style CoT reasoning loop (Yao et al., 2022) to predict repair actions based on interactive feedback from debuggers, while NExT focuses on tuning LLMs to reason with pre-existing execution information without intermediate feedback. Finally, as a related stream of research, self-improvement methods iteratively refine a model's code solutions using CoT reasoning over self-provided (Madaan et al., 2023) or test-driven feedback (Chen et al., 2023; Olausson et al., 2023). Instead of relying on high-level execution signals like error messages, NExT trains LLMs to reason with step-wise program traces. Our learnable rationales are also more flexible without following a predefined reasoning template. Besides, since traces already capture rich execution semantics, the resulting rationales could be more succinct and targeted to the downstream task (*e.g.,* explain bugs), without redundant reasoning steps to trace the program by the model itself to recover useful execution information.

Supervised CoT Reasoning LLMs can solve problems more accurately when instructed to work out the answer step by step in a *chain of thought* or a *scratchpad* (Nye et al., 2021; Rajani et al., 2019; Shwartz et al., 2020; Wei et al., 2022b). Improvements on this approach involve finetuning LLMs on chain-of-thought reasoning data. Such CoT data is either manually curated (Chung et al., 2022; Lightman et al., 2023; Longpre et al., 2023), or distilled from more capable teacher models (Fu et al., 2023; Gunasekar et al., 2023; Mitra et al., 2023; Mukherjee et al., 2023). Instead of relying on labeled or distilled data, NExT uses self-training to iteratively bootstrap a synthetic dataset of high-quality rationales with minimal manual annotation. Our work differs from previous work using bootstrapping (Hoffman et al., 2023; Zelikman et al., 2022) in the type of rationales and the

use of execution information; see §4.4 for more discussion. While we use the correctness of the program fix for filtering the rationales, which is reminiscent of outcome supervision; it is also possible to use process supervision with human annotations (Lightman et al., 2023; Uesato et al., 2022), or obtain such supervision automatically by estimating the quality of each step using Monte Carlo Tree Search (Wang et al., 2024b) and by identifying partially-correct program prefixes (Ni et al., 2022). Finally, existing research has investigated finetuning of LLMs to predict the execution information directly, such as predicting line-by-line execution traces (Nye et al., 2021), abstract runtime properties (Pei et al., 2023), or final output (Bieber et al., 2020; Zaremba and Sutskever, 2014). NExT addresses a different problem; instead of predicting the execution information, NExT takes it as given, and instead learns to discover flexible task-specific NL rationales that aid a downstream programming task.

4.12 Summary

In this chapter we present NExT, a self-training method to finetune LLMs to reason with program execution given traces. We demonstrate that PaLM 2-L trained using NExT yields high-quality natural language rationales and achieves stronger success rates on two program repair tasks. While NExT can potentially to applied to other domains, *e.g.,* improving coding capabilities in general, in our work, we use program repair as the primary task for experiments since it is a well-defined problem with clear evaluation criteria and real-world applications. Nevertheless, the ability to reason about program execution would benefit many more tasks beyond program repair, and we leave the evaluation and training for broader domains of coding applications as our main future work.

Chapter 5

Learning to Verify LM-Generated Code with Execution

In this chapter, we introduce LEVER, a method to improve the language-to-code generation performance of LLMs by learning a separate smaller model as a verifier to verify the program candidates sampled from the LLMs based on the natural language inputs. More specifically, LEVER learns to verify the program correctness based on the execution results of the candidate programs, and the verification scores from the verifiers will be used in combination with the generation probabilities from the LLMs to rerank the program samples to achieve higher accuracy. Experiments are conducted on 4 language-to-code generation tasks, covering domains such as text-to-SQL parsing, math reasoning and Python programming, and results show that LEVER significantly improves the performance of strong base LLMs such as Codex by 4.6% to 10.9%.[1]

5.1 Introduction

The ability of mapping natural language to executable code is the cornerstone of a variety AI applications such as database interfaces (Pasupat and Liang, 2015; Shi et al., 2020; Yu et al., 2018b), robotics control (Shridhar et al., 2020; Zhou et al., 2021) and virtual

1. The content of this chapter is directy adapted from *"LEVER: Learning to Verify Language-to-Code Generation with Execution"* by Ni *et. al* , 2023.

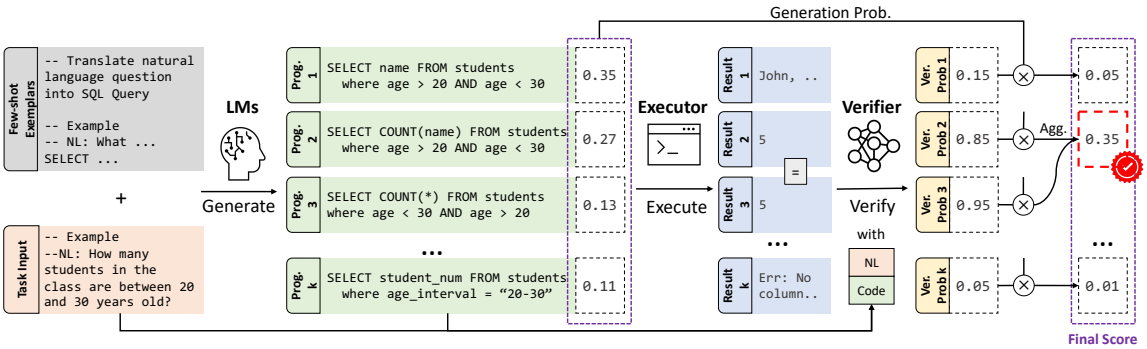


Figure 5.1: The illustration of LEVER using text-to-SQL as an example. It consists of three steps: 1) *Generation*: sample programs from LMs based on the task input and few-shot exemplars; 2) *Execution*: obtain the execution results with program executors; 3) *Verification*: using a learned verifier to output the probability of the program being correct based on the NL, program and execution results.

assistants (Agashe et al., 2019b; Lai et al., 2022). Recent advances on large language models (LLMs) (Brown et al., 2020; Chowdhery et al., 2022; Wei et al., 2021), especially those pre-trained on code (Chen et al., 2021a; Fried et al., 2022; Li et al., 2022b; Nijkamp et al., 2022), have shown great promise in such tasks with in-context few-shot learning (Chen et al., 2022a; Shi et al., 2022a; Zhang et al., 2022). Yet their performance is still far from perfect (Chen et al., 2021a). Considering the computation cost to finetune such models, it is appealing to explore ways to improve them without changing their parameters.

A key observation is that while LLMs struggles with precision in the few-shot setting, it often produces the correct output when enough samples are drawn. Previous work have shown that majority voting and filtering by test cases can significantly boost their performance when samples are drawn at scale (Austin et al., 2021a; Chen et al., 2021a; Li et al., 2022b). Shen et al. (2021) and Cobbe et al. (2021) further demonstrated the effectiveness of training a verifier and using the verification scores to rerank the candidate solutions for math world problems. Comparing to approaches that solely rely on execution consistency and error pruning, trained verifiers can make use of the rich semantic features in the model solutions, such as data types, value range, and variable attributes, which can be strong indicators of correctness of the programs. While Cobbe et al. (2021) and subsequent work (Kadavath et al., 2022; Li et al., 2022a) focus on verifying natural language solutions by LMs, a natural question is whether the same approach can be applied to program solutions.

In this chapter, we propose learning to verify (LEVER) language-to-code generation

by LLMs, with the help of execution. More specifically, we train a verifier that learns to distinguish and reject incorrect programs based on the joint representation of the natural language description, the program surface form and its execution result. We further combine the verification probability with the LLM generation probability and marginalize over programs with the same execution results. We use this aggregated probability as the reranking score and output the programs that execute to the most probable result.

We conduct extensive experiments on four different language-to-code benchmarks across domains of text-to-SQL semantic parsing, table QA, math reasoning and basic Python programming. Experiment results with three different LLMs show that LEVER consistently improves the execution accuracy of the generated programs. Notably, LEVER coupled with `code-davinci-002` improves over strong baselines that use execution error pruning by 4.6% to 10.9%, and achieves the new state-of-the-art results on all four benchmarks, without using task-specific model architecture or prompting methods. Ablation studies show that execution information is crucial for the verification and LEVER also yields non-trivial improvements in low-resource and weakly-supervised settings.

5.2 Approach

The key steps of LEVER is illustrated in Figure. 5.1. We now introduce the detailed formulation and learning of LEVER.

5.2.1 Language-to-Code Generation with LMs

The input for the language-to-code tasks typically consists of natural language (NL) description and optionally some programming context (*e.g.,* databases, assertions, etc). Given this task input x, a generation model $P(y|x)$ attempts to generate a program y which is later executed by an executor $\mathcal{E}(\cdot)$ to obtain the desired execution result $\mathcal{E}(y)$. For few-shot learning with large LMs, the generation is also often conditioned on a prompt$(x, \{(x_i, y_i)\}_{i<m})$, which is a function of the input x and a fixed set of m exemplars, $\{(x_i, y_i)\}_{i<m}$. Thus the

few-shot language-to-code generation with LMs can be formulated as:

$$P_{\mathbf{LM}}(y|x) = P(y|\,\text{prompt}(x, \{(x_i, y_i)\}_{i<m})) \tag{5.1}$$

Given input x, greedy search is typically used to output program with the (approximated) highest generation probability, *i.e.,* $\hat{y}_{\text{greedy}} \approx \arg\max_y P_{\mathbf{LM}}(y|x)$.

5.2.2 Reranking of Program Candidates

The key observation of our method is that while correct programs can often be found in the samples from $P_{\mathbf{LM}}(y|x)$, they are not always given the highest probabilities by the LM. The idea of discriminative reranking (Collins and Koo, 2005; Shen et al., 2004) is to learn a scoring function $R(x, \hat{y})$ that measures how likely $\hat{y}$ is the best output for input x. Thus given $R(\cdot)$, the reranker outputs the program with the highest reranking score among the set of candidates S:

$$\hat{y}_{\text{rerank}} = \arg\max_{\hat{y} \in S} R(x, \hat{y}) \tag{5.2}$$

Next we introduce how we adopt a trained verifier to verify and rerank program candidates sampled from LMs such that $\hat{y}_{\text{rerank}}$ is better than $\hat{y}_{\text{greedy}}$.

Program Sampling from LM. Given input x, instead of performing greedy search, we obtain k programs from $P_{\mathbf{LM}}(y|x)$ with temperature sampling, *i.e.,* $\{\hat{y}_i\}_{i=1}^{k} \sim P_{\mathbf{LM}}(y|x)$. As the same programs may be sampled more than once, we perform deduplication to form a set of n *unique* program candidates $S = \{\hat{y}_i\}_{i=1}^{n}$, where $n \leq k$. We choose to do sampling instead of beam search mainly for two reasons: 1) recent work suggests that beam search for code generation typically results in worse performance due to degenerated programs (Austin et al., 2021a; Zhang et al., 2022); and 2) beam search is not available or efficiently implemented for all LMs (*e.g.,* Codex) that we test on.

Verification with Execution. Traditionally, feature engineering is required to provide rich information as the input to the reranker (Collins and Koo, 2005; Shen et al., 2004; Yin and Neubig, 2019). But crafting such features is time-consuming and they typically do

not generalize across different domains. On the other hand, it is also challenging to rerank programs based on surface form as similar programs can have very different semantics. In this chapter, we use a simple concatenation of NL description x, candidate program $\hat{y}$ and a representation of its execution results $\mathcal{E}(\hat{y})$ as the input to the reranker, without the need for any feature engineering. Inspired by recent work (Cobbe et al., 2021; Li et al., 2022a), we parameterize our discriminative reranker as a verification (*i.e.,* binary classification) model $P_\theta(v|x, \hat{y}, \mathcal{E}(\hat{y}))$, where $v \in \{0, 1\}$. Thus given an NL input x and a candidate program $\hat{y} \in S$, we can obtain the *reranking probability* as the joint probability of generation and passing verification:

$$P_R(\hat{y}, v_{=1}|x) = P_{\mathbf{LM}}(\hat{y}|x) \cdot P_\theta(v_{=1}|x, \hat{y}, \mathcal{E}(\hat{y})) \tag{5.3}$$

Execution Result Aggregation. Since programs with the same semantics can have different surface forms, we further aggregate the reranking probability of programs in S that executes to the same result. In this way, we can relax the dependency on individual programs and replace with their execution results, yielding the final scoring function for reranking as:

$$R(x, \hat{y}) = P_R(\mathcal{E}(\hat{y}), v_{=1}|x) = \sum_{y \in S, \mathcal{E}(y)=\mathcal{E}(\hat{y})} P_R(y, v_{=1}|x)$$

Since there might be several programs that share the same execution result of the highest probability, we break tie randomly in this case when outputting the programs.

5.2.3 Learning the Verifiers

The previous sections described how to use a verifier to rerank the program candidates at inference time, here we introduce the learning of these verifiers.

Training Data Creation. For language-to-code datasets, each example is typically a triplet of (x, y^*, z^*), where $z^* = \mathcal{E}(y^*)$ is the gold execution result and y^* is the gold program. As annotating the programs may require domain experts, for some datasets, only z^* and no y^* is provided for learning. This is known as weakly-supervised learning (Artzi and Zettlemoyer, 2013; Cheng and Lapata, 2018; Goldman et al., 2018), and we show that LEVER applies to this setting as well. To gather training data, we obtain a set of n unique programs candidates

	Spider	**WikiTQ**	**GSM8k**	**MBPP**
Domain	Table QA	Table QA	Math QA	Basic Coding
Has program	✓	✓*	✗	✓
Target	SQL	SQL	Python	Python
# Train	7,000	11,321	5,968	378
# Dev	1,032	2,831	1,448	90
# Test	-	4,336	1,312	500

Table 5.1: Summary of the datasets used in this chapter. *: About 80% examples in WikiTableQuestions are annotated with SQL by Shi et al. (2020).

$\hat{S} = \{\hat{y}_i\}_{i=1}^{n}$ for each language input x in the training set, by first sampling k programs from $P_{\mathbf{LM}}(\hat{y}|x)$ and then remove all the duplicated programs, similar as inference time. Then for each program candidate $\hat{y} \in S$, we obtain its binary verification label by comparing the execution result $\hat{z} = \mathcal{E}(\hat{y})$ with the gold execution result z^*, *i.e.,* $v = \mathbb{1}(\hat{z} = z^*)$.[2] For the datasets equipped with the gold program y^*, we append $(x, y^*, z^*, v_{=1})$ as an additional verification training example, and this step is skipped for weakly-supervised datasets. In this way, for each NL input x, we have created a set of verification training examples $\{(x, \hat{y}_i, \hat{z}_i, v_i) \mid \hat{y}_i \in S\}$.

Learning Objective. Given this set of verification training examples, we formulate the loss for input x with the negative log-likelihood function, normalized by the number of program candidates

$$\mathcal{L}_\theta(x, S) = -\frac{1}{|S|} \cdot \sum_{\hat{y}_i \in S} \log P_\theta(v_i|x, \hat{y}_i, \hat{z}_i) \tag{5.4}$$

As result of this normalization, each x and each distinct $\hat{y}$ have equal weight.

5.3 Experimental Setup

5.3.1 Datasets

We conduct experiments on four language-to-code datasets across domains of semantic parsing, table QA, math reasoning and basic python programming. Detailed statistics and dataset-specific setups for these four datasets are shown in Table. 5.1.

2. Some tasks use test cases to check correctness, but we keep this notation for simplification.

▷ **Spider** (Yu et al., 2018b) is a semantic parsing dataset on generating SQL queries from natural language questions. With 7k parallel training data, it is also ideal for finetuning generators;

▷ **WikiTableQuestions (WikiTQ)** (Pasupat and Liang, 2015) is a table question answering dataset, for which we attempt to solve by generating and executing SQL queries over the source tables. We use the preprocessed tables from Shi et al. (2020) and adopt their annotated SQL queries for adding gold programs for the originally weakly-supervised dataset;

▷ **GSM8k** (Cobbe et al., 2021) is a benchmark for solving grade-school-level math word problems. Following previous work (Chen et al., 2022b; Chowdhery et al., 2022; Gao et al., 2022), we approach this benchmark by generating Python programs from questions in NL, which should produce the correct answer upon execution. The original dataset only has natural language and not program solutions, thus it is weakly-supervised for language-to-code;

▷ **MBPP** (Austin et al., 2021a) contains basic Python programming programs stated in natural language. Each example is equipped with 3 test cases to check the correctness of the programs. Following previous work (Shi et al., 2022a; Zhang et al., 2022), we use the first test case as part of the prompt for the model to generate correct function signatures and reserve the rest two to test for correctness;

5.3.2 Language Models

We evaluate LEVER with three different series of language models (LMs) :

▷ **Codex** (Chen et al., 2021a) is a series of LMs developed by OpenAI, we use the `code-davinci-002` engine accessed through the provided API;

▷ **InCoder** (Fried et al., 2022) is a series LMs trained on programming languages as Python and SQL. We mainly evaluate on the `InCoder-6B` version and use left-to-right generation though it also supports code infilling;

▷ **CodeGen** (Nijkamp et al., 2022) is a series of LMs up to 16B and we mainly evaluate the `CodeGen-16B-multi` version. Though CodeGen is not trained on SQL files, it still has non-trivial SQL generation ability since it may see SQL in the source files of other

programming languages as comments, etc.

5.3.3 Baselines and Evaluation Metric

Baselines. We consider the following heuristics for generating and ranking the program candidates from LLMs as baselines to improve upon:

▷ **Greedy**: Greedy decoding is used to generate a single program by selecting the most likely token at every step;

▷ **Maximum Likelihood (ML)**: Select the program with the highest generation probability (or normalized generation probability) from the program candidates [3];

▷ **ML + Error Pruning (EP)**: First prune out the programs with execution errors in the candidates, then select the program with the maximum likelihood;

▷ **Voting**: Take the majority vote on the execution results among the error-free programs, and select the most-voted execution result and its corresponding programs.

Evaluation metric. Following previous work, we use *execution accuracy* as the main evaluation metric, which measures the percentage of examples that yields the gold execution result or pass all test cases.

5.3.4 Implementation details.

We use the development set to choose the best verifier model. We use T5-base for Spider, T5-large for WikiTQ and MBPP, and RoBERTa-large for GSM8k as the base LM for the verifiers, unless specified otherwise. More discussion and results with different base models can be found in § 5.7.1. Given the verification input, we train T5 models to output token "yes" or "no" representing the positive/negative label, and we take the probability of generating the "yes" token as the verification probability during inference. For RoBERTa, we add a linear layer on top as standard practice for sequence classification with encoder-only models. More detailed implementation details can be found in § 5.6.

3. Whether to use normalized probability or not is determined empirically. This distinction is also consistent with how generation probability is used in verification in Equation. 5.3.

Methods	Dev
Previous Work without Finetuning	
Rajkumar et al. (2022)	67.0
MBR-Exec (Shi et al., 2022a)	75.2
Coder-Reviewer (Zhang et al., 2022)	74.5
Previous Work with Finetuning	
T5-3B (Xie et al., 2022)	71.8
PICARD (Scholak et al., 2021)	75.5
RASAT (Qi et al., 2022)	80.5
This Work with code-davinci-002	
Greedy	75.3
ML + EP	77.3
LEVER	**81.9**$_{\pm 0.1}$

Table 5.2: Execution accuracy on the Spider dataset. Standard deviation is calculated over three runs with different random seeds (same for the following tables when std is presented).

Methods	Dev	Test
Previous Work without Finetuning		
Codex QA* (Cheng et al., 2022)	50.5	48.7
Codex SQL† (Cheng et al., 2022)	60.2	61.1
Codex Binder† (Cheng et al., 2022)	**65.0**	64.6
Previous Work with Finetuning		
TaPEX* (Liu et al., 2021)	60.4	59.1
TaCube (Zhou et al., 2022b)	61.1	61.3
OmniTab* (Jiang et al., 2022)	-	63.3
This Work with code-davinci-002		
Greedy	49.6	53.0
ML + EP	52.7	54.9
LEVER	64.6$_{\pm 0.2}$	**65.8**$_{\pm 0.2}$

Table 5.3: Execution accuracy on the WikiTQ dataset. *: modeled as end-to-end QA without using programs; †: an LM-enhanced version SQL/Python programs are used.

5.4 Main Results

Here we present our main results by comparing LEVER with aforementioned baselines as well as previous work on the four datasets we study. We also perform ablation study on LEVER in this section.

5.4.1 Effectiveness of Lever.

Here we show the performance of LEVER coupled with Codex and compare with the best finetuning and few-shot performances from previous work for Spider (Table. 5.2), WikiTQ (Table. 5.3), GSM8k (Table. 5.4) and MBPP (Table. 5.5). In addition, we also evaluate

Methods	Dev	Test
Previous Work without Finetuning		
PAL (Gao et al., 2022)	-	72.0
Codex + SC† (Wang et al., 2022b)	-	78.0
PoT-SC (Chen et al., 2022b)	-	80.0
Previous Work with Finetuning		
Neo-2.7B + SS (Ni et al., 2022)	20.7	19.5
Neo-1.3B + SC (Welleck et al., 2022)	-	24.2
DiVeRSe*† (Li et al., 2022a)	-	83.2
This Work with codex-davinci-002		
Greedy	68.1	67.2
ML + EP	72.1	72.6
LEVER	**84.1**$_{\pm 0.2}$	**84.5**$_{\pm 0.3}$

Table 5.4: Execution accuracy on the GSM8k dataset. *: model finetuned on top of codex (similar to LEVER); †: natural language solutions are used instead of programs.

LEVER with InCoder and CodeGen models on the Spider and GSM8k datasets, and the results are shown in Table. 5.6. From the results, we can see that LEVER consistently improves the performance of LLMs on all tasks, yielding improvements of 6.6% (Spider) to 17.3% (WikiTQ) over the greedy decoding baselines for Codex. For weaker models as InCoder and CodeGen, such improvements can be up to 30.0% for Spider and 15.0% for GSM8k. Moreover, LEVER combined with Codex also achieves new state-of-the-art results on all four benchmarks, with improvements from 1.2% (WikiTQ) to 2.0% (MBPP). On the challenging text-to-SQL dataset Spider, where the previous best method is by finetuning a 3B parameter model with tasks-specific architecture (Qi et al., 2022), with Codex + LEVER, best results are achieved by finetuning only a T5-base model on top of the Codex few-shot outputs. LEVER also improves the previous best results on Spider for InCoder and CodeGen, by 13.2% and 20.6%, respectively. As LEVER is a simple method that combines the power of few-shot LLMs and finetuned verifiers, it can also potentially benefit from better prompting methods (Cheng et al., 2022; Li et al., 2022a) or model architectures (Qi et al., 2022; Wang et al., 2020) from previous work, for which we will leave as future work.

5.4.2 Ablations with Lever

We perform ablation study for LEVER with Codex and compare with the baselines mentioned in § 5.3.3, and the results are shown in Figure. 5.2. The same ablations are conducted for

Methods	Dev	Test
Previous Work without Finetuning		
MBR-Exec (Shi et al., 2022a)	-	63.0
Reviewer (Zhang et al., 2022)	-	66.9
This Work with codex-davinci-002		
Greedy	61.1	62.0
ML + EP	62.2	60.2
LEVER	**75.4**$_{\pm 0.7}$	**68.9**$_{\pm 0.4}$

Table 5.5: Execution accuracy on the MBPP dataset. No previous finetuning work we found is comparable.

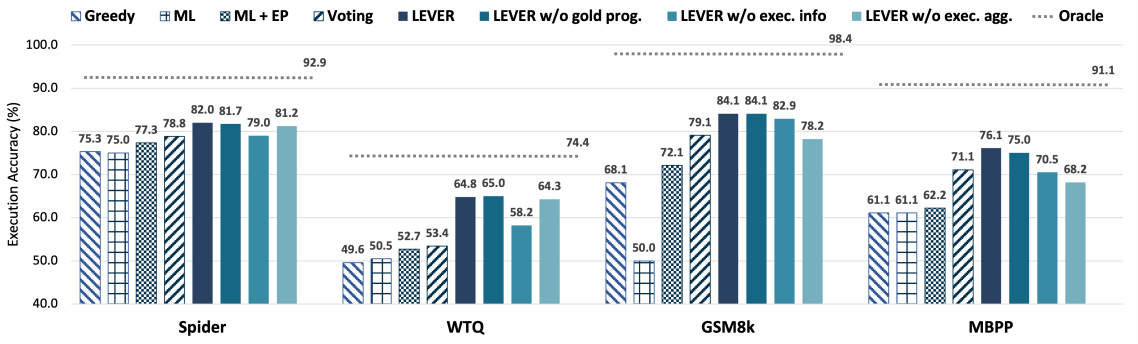


Figure 5.2: Comparison of LEVER with Codex002 baseline methods. All LEVER results are in solid bars.

InCoder and CodeGen with results in Table. 5.6.

Execution information. From Figure. 5.2, we can notice that the performance drops considerably on all four benchmarks when execution information is removed from the input of the verifiers, proving that execution information is essential to the success of learning these verifiers for reranking. Moreover, we can also observe that the effect of execution information varies for different LLMs and datasets, as the performance gaps are larger for weaker LLMs and harder domains without execution information. This suggests that the verifiers heavily rely on the execution information to make decisions in these more difficult settings. Moreover, we find that LEVER typically outperforms the ML+EP baseline, indicating what the verifiers are able to learn from the execution results are beyond simply execution errors. More detailed analysis of this is in Figure. 5.5.

Execution aggregation. Aggregating the programs with the same execution result is a simply and popular method (Chen et al., 2022b; Cheng et al., 2022). In our experiments, we

Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
Previous work:				
MBR-Exec	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
Baselines:				
Greedy	24.1	3.1	24.6	7.1
ML	33.7	3.8	31.2	9.6
ML + EP	41.2	4.4	37.7	11.4
Voting	37.4	5.9	37.1	14.2
Lever	54.1	**11.9**	51.0	**22.1**
− gold prog.	53.4	-	**52.3**	-
− exec. info	48.5	5.6	43.0	13.4
− exec. agg.	**54.7**	10.6	51.6	18.3
Oracle	71.6	48.0	68.6	61.4

Table 5.6: Results with more LLMs, evaluated on the dev set with T5-base as the verifier. Previous work numbers taken from Zhang et al. (2022).

find execution aggregating works well with Lever on the benchmarks that uses Python as programs, and no benefit is observed for SQL datasets. We think this is expected as Python grammar is much more flexible, which is more likely to produce semantically equivalent programs in different surface forms. However, for the less-flexible SQL, enable execution aggregation makes it more susceptible to spurious programs (Cheng and Lapata, 2018; Misra et al., 2018).

Weakly-supervised settings. Here we compare the performance of Lever under fully- and weakly-supervised settings. From Figure. 5.2 and Table. 5.6, we can see that the performance of Lever is generally preserved when gold programs are not available for learning, with the performance gap up to 1.1%. This suggests that Lever also works well under weakly-supervised settings, and it is a nice property as domain experts are typically required to write gold programs which can be very expensive (Cheng and Lapata, 2018).

5.5 Analysis

Here we present more analysis on Lever to better understand: 1) How data efficient is the learning of verification (§ 5.5.1); 2) What are the most common reasons for Lever to succeed or fail (§ 5.5.2).

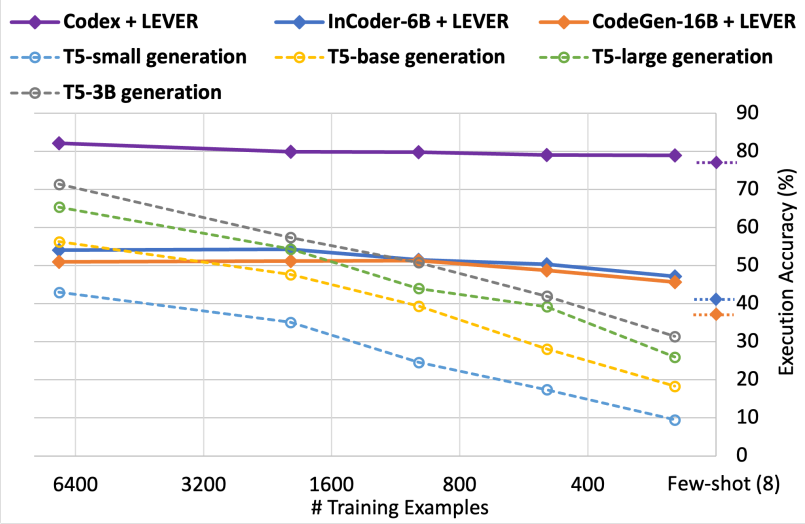


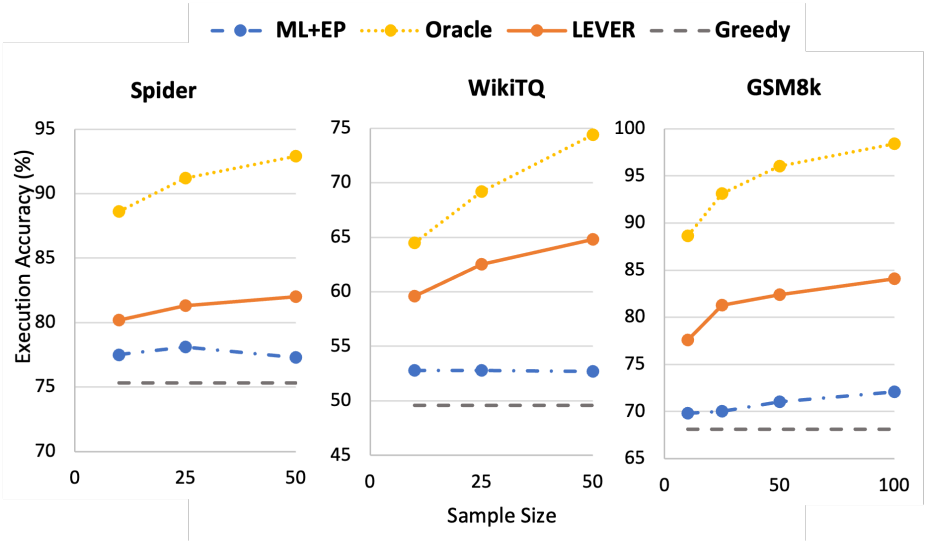
Figure 5.3: Verification vs. generation performance when decreasing the number of training examples for Spider. Data markers on the y-axis denote the ML+EP baseline. WikiTQ and GSM8k results can be found in Figure. 5.6 in the Appendix.

5.5.1 Data Efficiency

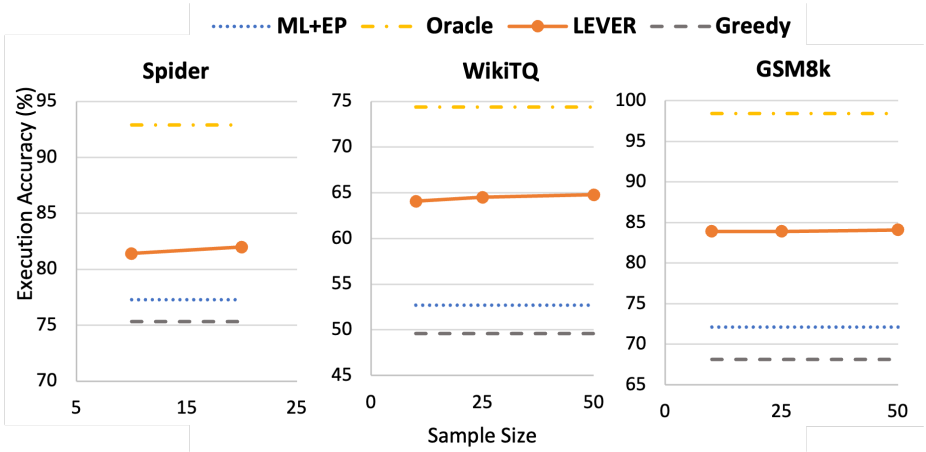
The amount of training data for LEVER is dominated by two factors: the number of training examples in the original language-to-code dataset, and sample size per example when sampling from LLMs, the later of which is also relevant during inference. Here we study how data efficient is the learning of LEVER.

Training data scaling. We show how the performance of LEVER changes when less training data is available in Figure. 5.3. The improvements with LEVER over LLMs are still consistent even when only 250 examples are given, with improvements ranging from 1.7% to 10.0% over different datasets and LLMs. This suggests that LEVER can work under few-resource settings. Moreover, the trend also varies for different datasets and LMs, for example, when using Codex as the LM, the performance of LEVER drops by 6.4% for WikiTQ and only 3.2% for Spider. However, also on Spider, the performance is lowered by 6.9% and 5.3% for InCoder and Codegen. This suggests that having more training examples for LEVER has larger effect for harder datasets and weaker LMs.

Finetuning for generation. Since Spider have enough parallel training data, most previous work, including previous SoTA, performs finetuning on Spider. With Figure. 5.3,



(a) Ablation on sample size at inference time for LEVER, while sample size at training time is fixed as in Table. 5.1.



(b) Performance with different number of programs to sample per example for training the verifiers. Sample size at inference time is fixed as in Table. 5.1, thus baseline performances do not change.

Figure 5.4: How sample size during training and inference time affects the performance, using Codex as the LLM.

we compare the performance of LEVER with the T5 models being directly finetuned for generation given the same number of training examples. While verification can be learned with only hundreds of examples, the performance of finetuned T5 models drastically drops when less training examples are available. As an example, for 500 examples, a T5-base verifier on InCoder/CodeGen outperforms a finetuned T5-3B generator by $\sim 7\%$.

Target LM & ML+EP Baseline		Source LM (% Positive Labels)		
		Codex (64.0%)	InCoder (9.2%)	CodeGen (8.6%)
Codex	77.3	82.0 (+4.7)	81.7 (+4.4)	80.8 (+3.5)
InCoder	41.2	46.4 (+5.2)	54.1 (+12.9)	47.6 (+6.4)
CodeGen	37.7	44.7 (+7.0)	48.9 (+11.2)	51.0 (+13.3)

(a) Between the LMs transfer results on Spider.

Target LM & ML+EP Baseline		Source LM (% Positive Labels)		
		Codex (53.4%)	InCoder (2.3%)	CodeGen (5.0%)
Codex	72.1	83.7 (+11.6)	70.0 (-2.1)	71.9 (-0.2)
InCoder	4.3	8.3 (+4.0)	11.9 (+7.6)	12.3 (+8.0)
CodeGen	9.6	18.4 (+8.8)	20.7 (+11.1)	22.1 (+12.5)

(b) Between the LMs transfer results on GSM8k.

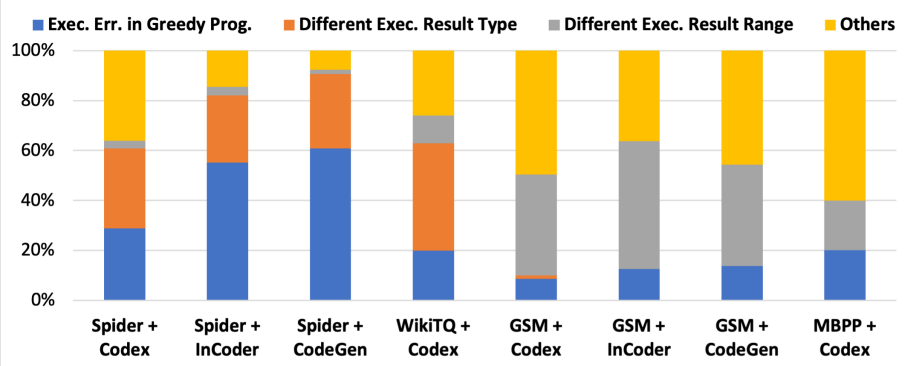
Table 5.7: Execution accuracy of training verifiers on the programs sampled from source LM and apply to the target LM. The best and second best performance per row is highlighted accordingly.

Sample Size. Since drawing samples from LMs in may be costly computational-wise, here we study the how sample size during training and inference time affects the performance. As we can see from Figure. 5.4a, during inference time, when lowering the sample size from 50 to 10 programs per example, the performance of LEVER drops by 1.8% (Spider) to 5.2% (WikiTQ). This indicates that the LEVER is sensitive to the sample size at inference time, which is expected as it also greatly affects oracle results (*i.e.,* the upper-bound for reranking). In comparison, Figure. 5.4b shows that LEVER is highly insensitive to the sample size for providing training data, with the performance gap all below 1% for the three datasets. Overall, the results show that a higher sampling budget helps more at test time.

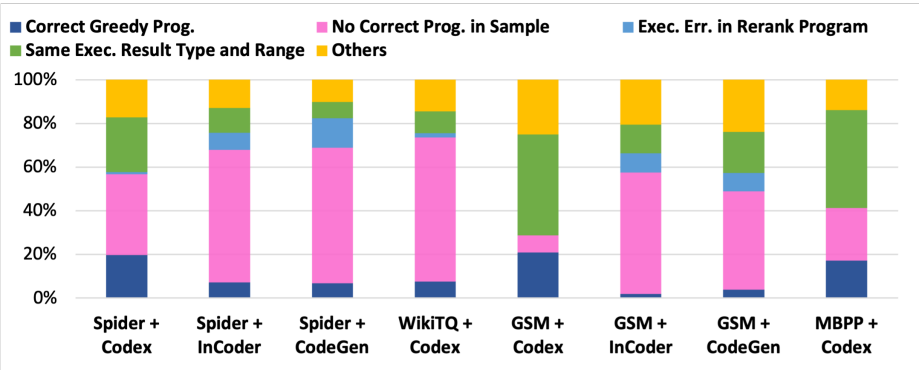
Transfer Learning between LMs. One other way to avoid the cost of sampling from LMs is to train verifiers using samples from one LM and directly apply to the programs sampled from a different LM, *i.e.,* between LM transfer, and we show the results of such on Spider and GSM8k in Table. 5.7. From the results, we can first observe that LEVER still non-trivially improves the baseline performance most of the time, with the exception of transferring from InCoder and CodeGen to Codex on the GSM8k dataset. This suggests that the knowledge learned by the verifiers are generalizable to different LM outputs. Moreover,

we can see that the transfer typically works better when the percentage of positive labels are closer, as the transfer is more successful between the InCoder and CodeGen models than that with Codex. These results show between-LM transfer as an interesting way to reduce the training data need for LEVER.

5.5.2 Quantitative Analysis



(a) When LEVER reranks a correct program first when the greedy decoded program is incorrect.



(b) When LEVER fails to rank correct programs to the top.

Figure 5.5: Quantitative analysis on when LEVER succeeds and fails to improve LMs over the greedy decoding.

In Figure. 5.5, we present a quantitative analysis on the type of reasons of why LEVER successfully or failed to improve the performance of LMs. From the results, we can see that when LEVER reranks a correct program to replace the greedy decoding output, it is likely that the execution results provide crucial information such as execution errors, variable type and range. This is consistent with our findings in § 5.4.2 about the importance of execution information for LEVER. It is also worth noticing that there are cases when LEVER is still

able to rerank the correct program when the error-free execution results are of the same type and range with the greedy program, *i.e.,* in "others" category. Our hypothesis is that this is when the program itself becomes the main feature for the verifiers to exploit. In addition, when LEVER fails to rank correct programs to the top, the most common reason is that no correct program can be found in the samples (*i.e.,* upper-bound is reached), which is especially the case for weaker LMs. The second most common reason for LEVER to fail is that the execution results of the incorrect program upon reranking has the same type and range as the correct program in the samples. In this case, execution results do not provide rich information for the verifiers thus LEVER fails to improve LMs.

5.6 Detailed Experiment Setup

Down-sampling during training. As shown in Equation. 5.4, the training loss per example is computed by averaging over all the program samples for that example. However, this could be problematic when sample size gets large (up to 100 in our experiments) as they may not be able to fit into the GPU memory at once. Thus during implementation, we down-sample the programs used for learning per example in each iteration. The down-sampling happens at the beginning of every epoch of training so the verifiers are able to see different programs each epoch.

Few-shot exemplars. The numbers of few-shot exemplars to include in the prompt for different datasets are shown in Table. 5.1. All the exemplars are randomly sampled from the training set, with the exception of MBPP, where we use the 3 examples designated as few-shot exemplars in the original dataset. Full prompts used for each dataset are shown in § 5.10.

Sampling details. We use temperature sampling to obtain program candidates given the different input formats and sampling budgets as described in Table. 5.8. We set the temperature as $T = 0.6$ for Codex and $T = 0.8$ for InCoder and CodeGen by referring to the original papers for optimal sampling temperatures of higher pass@k given the sample sizes. Ablation studies on sampling budget is shown in Figure. 5.4.

	Spider	WTQ	GSM8k	MBPP
Few-shot Generation Settings				
Input Format	Schema + NL	Schema + NL	NL	Assertion + NL
# Shots	8‡	8	8	3
# Samples (train / test)	20/50†	50/50	50/100	100/100
Generation Length	128	128	256	256
Verification Settings				
Input Format	NL+ SQL+ Exec.	NL+ SQL+ Exec.	NL+ Prog.+ Exec.	NL+ Prog.+ Exec.
Normalize Gen. Prob.	No	No	Yes	Yes

Table 5.8: Hyperparameters for few-shot generation and learning the verifiers. †: 50/100 for InCoder and CodeGen for improving the upper-bound; ‡: only the first 2 of the 8 exemplars are used for InCoder and CodeGen due to limits of context length and hardware.

Presenting execution information. The input to the verifier is a concatenation of the natural language input, the candidate program and its execution information. For Spider and WikiTQ, we present the execution information simply as the linearized resulting table of the SQL execution. For GSM8k, the execution information is presented as the value of the "answer" variable after executing the program. For MBPP, we use the type and value (casted to string) returned by the function as the execution information. All execution errors will be represented as "`ERROR: [reason]`", such as "`ERROR: Time out`". Examples of these verifier inputs for different datasets can be found in Table. 5.12.

Dataset-specific setups. The detailed experiment setups for specific datasets are shown as Table. 5.8.

5.7 Additional Results

5.7.1 Ablation on Base LMs for Verification

In this chapter, we treat the choice of base models for the verifiers as a hyperparameter and use the best performing model for further experiments. Here we show the performance of all the base models we attempted on the four datasets, with results in Table. 5.9.

Base LMs	Spider	WTQ	GSM8k	MBPP
T5-base	**82.0**	64.8	82.4	76.8
T5-large	81.9	**65.0**	82.5	**77.3**
T5-3B	83.1	64.7	**84.4**	-
RoBERTa-large	-	64.3	**84.4**	-

Table 5.9: Ablations on using different base models for the verifiers. -: base LM not tested on this dataset.

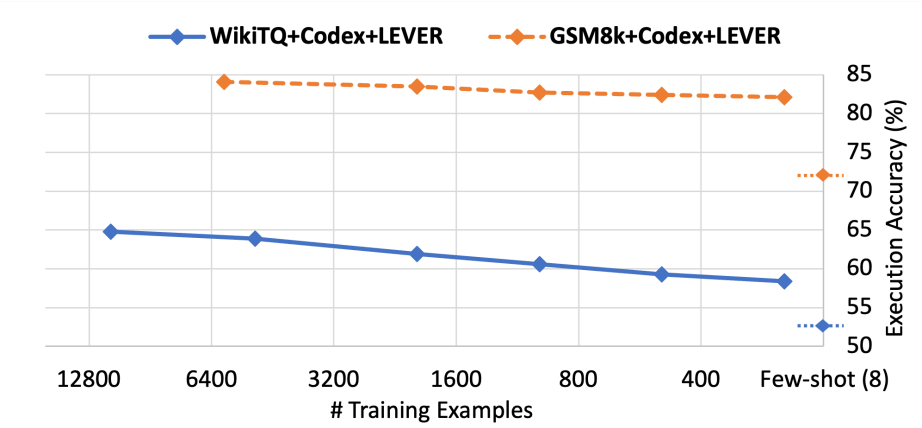


Figure 5.6: Ablation on number of training examples for Codex+LEVER on the WTQ and GSM8k datasets. Data markers on the y-axis denote the ML+EP performances as baselines. T5-base is used for LEVER.

5.7.2 Training Example Scaling for WTQ and GSM8k

Due to space limit, we are only able to show the ablation in number of training example for Spider. Here in Figure. 5.6, we show the results for WikiTQ and GSM8k as well. From the results, we can see that the learning of LEVER is also very data efficient on those two benchmarks, as non-trivial improvements can be observed even when only 250 training examples are given.

5.7.3 WikiTQ Results with the Official Evaluator

Following Cheng et al. (2022), we fix the official evaluator of WikiTQ by normalizing units, Boolean values, etc. Here we also report the performance of LEVER with previous work based on the official WikiTQ evaluator in Table. 5.10. From the results, we can see that LEVER still presents the state-of-the-art result under this setting.

Methods	Dev	Test
Previous Work without Finetuning		
Codex QA‡ (Cheng et al., 2022)	49.3	47.6
Codex SQL‡ (Cheng et al., 2022)	57.6	55.1
Codex Binder‡ (Cheng et al., 2022)	**62.6**	61.9
Previous Work with Finetuning		
TaPEX* (Liu et al., 2021)	57.0	57.5
TaCube (Zhou et al., 2022b)	59.7	59.6
OmniTab (Jiang et al., 2022)	-	**62.8**
This Work Using code-davinci-002		
Greedy	47.2	50.9
ML	48.3	50.9
ML + EP	50.1	52.5
Voting	50.6	53.6
LEVER	$61.1_{\pm 0.2}$	$\mathbf{62.9}_{\pm 0.2}$
Oracle	70.9	74.6

Table 5.10: Execution accuracy on the WTQ dataset with the official WTQ executor. ‡: a normalizer to recognize date is added to the official executor.

5.7.4 Case Study

Here we give some concrete examples to illustrate how LEVER work and when does it fail in Table. 5.11. In the first example from the Spider dataset, we can see that program candidate $\hat{y}_2$ selects from the wrong table, which results in an execution error. This is easily detected by the verifier thus put a low verification probability on such program. Meanwhile, the execution result $\hat{z}_1$ from program $\hat{y}_1$ seems much more likely to be the answer of the question to the verifier. In the second example from WikiTQ, however, the execution results $\hat{z}_1$ and $\hat{z}_2$ do not provide clear information as they are both county names. In this case, the verifier does not possess much more meaningful information than the generator, thus not able to identify the incorrect program.

5.8 Related Work

Language-to-Code Generation. Translating natural language to code is a long-standing challenge through all eras of artificial intelligence, including rule-based systems (Templeton and Burger, 1983; Woods, 1973a), structured prediction (Gulwani and Marron, 2014; Zelle and Mooney, 1996a; Zettlemoyer and Collins, 2005) and deep learning (Dong and Lapata,

Spider Example

Question x: Find the total ranking points for each player and their first name.
Program $\hat{y}_1$ (correct):
`select first_name, sum(ranking_points) from players join rankings on player.player_id = rankings.player_id group by first_name`
Program $\hat{y}_2$ (incorrect):
`select first_name, sum(ranking_points) from rankings join players on rankings.player_id = players.player_id group by player_id`
Execution info $\hat{z}_1$:
`Aastha | 68; Abbi | 304; Abbie | ...`
Execution info $\hat{z}_2$:
`ERROR: not column named ...`

WikiTQ Example

Question x: When ranking the counties from first to last in terms of median family income, the first would be?
Program $\hat{y}_1$ (incorrect):
`select county from main_table order by median_family_income_number limit 1`
Program $\hat{y}_2$ (correct):
`select county from main_table order by median_family_income_number desc limit 1`
Execution info $\hat{z}_1$: `county | jefferson`
Execution info $\hat{z}_2$: `county | sanders`

Table 5.11: Case study for the WikiTQ and Spider datasets. Program $\hat{y}_1$ is **ranked above** program $\hat{y}_2$ in both examples. The main differences in the SQL programs that lead to error are highlighted.

2016; Lin et al., 2017b; Rabinovich et al., 2017; Xiao et al., 2016; Zhong et al., 2017a). Recently, pre-trained code language models (Chen et al., 2021a; Fried et al., 2022; Nijkamp et al., 2022; OpenAI, 2022; Wang et al., 2021) have demonstrated surprisingly strong performance in this problem across programming languages (Austin et al., 2021a; Cobbe et al., 2021; Li et al., 2022b; Lin et al., 2018; Yu et al., 2018b). A number of approaches were proposed to refine LM sample selection, including test case execution (Li et al., 2022b), cross-sample similarity (Chen et al., 2021a; Li et al., 2022b; Shi et al., 2022a) and maximum mutual information (Zhang et al., 2022) based filtering. Our work proposes a learnable verification module to judge the sample output of LMs to further improve their performance.

Code Generation with Execution. Previous code generation work have exploited execution results in different ways. Weakly-supervised learning approaches (Berant et al., 2013; Guu et al., 2017a; Pasupat and Liang, 2015) model programs as latent variables and use execution results to derive the supervision signal. Intermediate execution results were used to guide program search at both training (Chen et al., 2019, 2021c) and inference time (Wang et al., 2018b). When sampling at scale, majority voting based on the execution results has

SPIDER/WIKITQ: question + SQL + linearized result table

Input:
```
-- question:  List the name, born state and age of the heads of departments ordered by
age.|
-- SQL:|select name, born_state, age from head join management on head.head_id =
management.head_id order by age|
-- exec result:|/*| name born_state age| Dudley Hart California 52.0| Jeff Maggert Delaware
53.0|Franklin Langham Connecticut 67.0| Billy Mayfair California 69.0| K. J. Choi Alabama
69.0|*/
```
Output: no

GSM8K: question + idiomatic program + answer variable

Input:
```
Carly recently graduated and is looking for work in a field she studied for.  She sent 200
job applications to companies in her state, and twice that number to companies in other
states.  Calculate the total number of job applications she has sent so far.  |
n_job_apps_in_state = 200
n_job_apps_out_of_state = n_job_apps_in_state * 2
answer = n_job_apps_in_state + n_job_apps_out_of_state |
'answer':  600
```
Output: yes

MBPP: task description + function + return type & value

Input:
```
# description
Write a function to find the n-th power of individual elements in a list using lambda
function.

# program
def nth_nums(nums,n):
    result_list = list(map(lambda x:  x ** n, nums))
    return (result_list)

# execution
# return:  (list)=[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
# return:  (list)=[1000, 8000, 27000]
# return:  (list)=[248832, 759375]
```
Output: yes

Table 5.12: Examples of verifier inputs on the datasets. Newlines are manually inserted for better display.

been shown effective for candidate selection (Cobbe et al., 2021; Li et al., 2022b). Shi et al. (2022a) generalizes this principle by selecting samples that have the maximum concensus with other samples in the execution results. We propose to train a verification model to judge the correctness of code generation taking the execution results into account.

Learning to Verify. Previous work have shown the effectiveness of learned verifiers for candidate filtering in domains such as math QA (Cobbe et al., 2021; Shen et al., 2021) and commonsense QA (Li et al., 2022a), where the solution is mostly described in natural language.

Although it is more common to train the verifiers independently from the generator (Cobbe et al., 2021; Li et al., 2022a), Shen et al. (2021) jointly fine-tuned both at the same time. Different base LMs were used as the verifiers. Cobbe et al. (2021) uses GPT-3 (Brown et al., 2020) while Li et al. (2022a) uses DeBERTa (He et al., 2020). In addition, Kadavath et al. (2022) shows that large LMs can self-verify their output in a few-shot setting. In comparison, the setting of LEVER is closer to Li et al. (2022a) as we train the verifier separately and use a much smaller LM for it (approximately 0.5% of the generator parameter size). We report the first set of comprehensive evaluation on language-to-code tasks, making use of the program execution results. Kadavath et al. (2022) also reported self-verification results on HumanEval. However, their approach does not leverage execution results.

5.9 Summary

In this chapter, we introduce LEVER, a simple approach for improving LMs on language-to-code tasks, by learning to verify the generated programs with their execution results. Experiments on four language-to-code tasks show that LEVER consistently improves the performance of LMs, and achieves the new state-of-the-art results on all benchmarks. Ablation studies suggest that execution information is crucial for verification, and further analysis shows that the learning of LEVER is data efficient and the learnt knowledge is generalizable across different LMs.

5.10 Appendix – Prompts for Few-shot Generation

Finally, we append the full prompts we used for few-shot prompting the LMs for Spider (Table. 5.13, Table. 5.14, Table. 5.15), WikiTQ (Table. 5.16, Table. 5.17), GSM8k (Table. 5.18), and MBPP (Table. 5.20).

```
-- Translate natural language questions into SQL queries.

-- Example:

-- Database game_injury:
--  Table stadium: id, name, Home_Games, Average_Attendance, Total_Attendance, Capacity_Percentage
--  Table game: stadium_id, id, Season, Date, Home_team, Away_team, Score, Competition
--  Table injury_accident: game_id, id, Player, Injury, Number_of_matches, Source
-- Question: How many distinct kinds of injuries happened after season 2010?
-- SQL:
SELECT count(DISTINCT T1.Injury) FROM injury_accident AS T1 JOIN game AS T2 ON T1.game_id  =  T2.id
WHERE T2.Season  >  2010

-- Example:

-- Database farm:
--  Table city: City_ID, Official_Name, Status, Area_km_2, Population, Census_Ranking
--  Table farm: Farm_ID, Year, Total_Horses, Working_Horses, Total_Cattle, Oxen, Bulls, Cows, Pigs,
Sheep_and_Goats
--  Table farm_competition: Competition_ID, Year, Theme, Host_city_ID, Hosts
--  Table competition_record: Competition_ID, Farm_ID, Rank
-- Question: Return the hosts of competitions for which the theme is not Aliens?
-- SQL:
SELECT Hosts FROM farm_competition WHERE Theme !=  'Aliens'
```

Table 5.13: The prompt we use for the Spider dataset for few-shot generation with LMs. Only the first 2 exemplars are shown here, which is also the only two used for InCoder/CodeGen due to limits of model length and computation. 8 total exemplars are used for Codex, and the rest are shown in Table. 5.14 and Table. 5.15.

```
-- Example:

-- Database school_finance:
--  Table School: School_id, School_name, Location, Mascot, Enrollment, IHSAA_Class,
IHSAA_Football_Class, County
--  Table budget: School_id, Year, Budgeted, total_budget_percent_budgeted, Invested,
total_budget_percent_invested, Budget_invested_percent
--  Table endowment: endowment_id, School_id, donator_name, amount
-- Question: Show the average, maximum, minimum enrollment of all schools.
-- SQL:
SELECT avg(Enrollment) ,  max(Enrollment) ,  min(Enrollment) FROM School

-- Example:

-- Database cre_Docs_and_Epenses:
--  Table Ref_Document_Types: Document_Type_Code, Document_Type_Name, Document_Type_Description
--  Table Ref_Budget_Codes: Budget_Type_Code, Budget_Type_Description
--  Table Projects: Project_ID, Project_Details
--  Table Documents: Document_ID, Document_Type_Code, Project_ID, Document_Date, Document_Name,
Document_Description, Other_Details
--  Table Statements: Statement_ID, Statement_Details
--  Table Documents_with_Expenses: Document_ID, Budget_Type_Code, Document_Details
--  Table Accounts: Account_ID, Statement_ID, Account_Details
-- Question: Return the ids and details corresponding to projects for which there are more than two
documents.
-- SQL:
SELECT T1.Project_ID ,  T1.Project_Details FROM Projects AS T1 JOIN Documents AS T2 ON T1.Project_ID
=  T2.Project_ID GROUP BY T1.Project_ID HAVING count(*)  >  2

-- Example:

-- Database local_govt_in_alabama:
--  Table Services: Service_ID, Service_Type_Code
--  Table Participants: Participant_ID, Participant_Type_Code, Participant_Details
--  Table Events: Event_ID, Service_ID, Event_Details
--  Table Participants_in_Events: Event_ID, Participant_ID
-- Question: List the type of the services in alphabetical order.
-- SQL:
SELECT Service_Type_Code FROM Services ORDER BY Service_Type_Code

-- Example:

-- Database cre_Theme_park:
--  Table Ref_Hotel_Star_Ratings: star_rating_code, star_rating_description
--  Table Locations: Location_ID, Location_Name, Address, Other_Details
--  Table Ref_Attraction_Types: Attraction_Type_Code, Attraction_Type_Description
--  Table Visitors: Tourist_ID, Tourist_Details
--  Table Features: Feature_ID, Feature_Details
--  Table Hotels: hotel_id, star_rating_code, pets_allowed_yn, price_range, other_hotel_details
--  Table Tourist_Attractions: Tourist_Attraction_ID, Attraction_Type_Code, Location_ID,
How_to_Get_There, Name, Description, Opening_Hours, Other_Details
--  Table Street_Markets: Market_ID, Market_Details
--  Table Shops: Shop_ID, Shop_Details
--  Table Museums: Museum_ID, Museum_Details
--  Table Royal_Family: Royal_Family_ID, Royal_Family_Details
--  Table Theme_Parks: Theme_Park_ID, Theme_Park_Details
--  Table Visits: Visit_ID, Tourist_Attraction_ID, Tourist_ID, Visit_Date, Visit_Details
--  Table Photos: Photo_ID, Tourist_Attraction_ID, Name, Description, Filename, Other_Details
--  Table Staff: Staff_ID, Tourist_Attraction_ID, Name, Other_Details
--  Table Tourist_Attraction_Features: Tourist_Attraction_ID, Feature_ID
-- Question: Show the average price range of hotels that have 5 star ratings and allow pets.
-- SQL:
SELECT avg(price_range) FROM Hotels WHERE star_rating_code  =  "5" AND pets_allowed_yn  =  1
```

Table 5.14: The prompt we use for the Spider dataset for few-shot generation with LMs (Part 2), continued from Table. 5.13.

```
-- Example:

-- Database insurance_fnol:
--  Table Customers: Customer_ID, Customer_name
--  Table Services: Service_ID, Service_name
--  Table Available_Policies: Policy_ID, policy_type_code, Customer_Phone
--  Table Customers_Policies: Customer_ID, Policy_ID, Date_Opened, Date_Closed
--  Table First_Notification_of_Loss: FNOL_ID, Customer_ID, Policy_ID, Service_ID
--  Table Claims: Claim_ID, FNOL_ID, Effective_Date
--  Table Settlements: Settlement_ID, Claim_ID, Effective_Date, Settlement_Amount
-- Question: Find all the phone numbers.
-- SQL:
SELECT Customer_Phone FROM available_policies

-- Example:

-- Database cre_Theme_park:
--  Table Ref_Hotel_Star_Ratings: star_rating_code, star_rating_description
--  Table Locations: Location_ID, Location_Name, Address, Other_Details
--  Table Ref_Attraction_Types: Attraction_Type_Code, Attraction_Type_Description
--  Table Visitors: Tourist_ID, Tourist_Details
--  Table Features: Feature_ID, Feature_Details
--  Table Hotels: hotel_id, star_rating_code, pets_allowed_yn, price_range, other_hotel_details
--  Table Tourist_Attractions: Tourist_Attraction_ID, Attraction_Type_Code, Location_ID,
How_to_Get_There, Name, Description, Opening_Hours, Other_Details
--  Table Street_Markets: Market_ID, Market_Details
--  Table Shops: Shop_ID, Shop_Details
--  Table Museums: Museum_ID, Museum_Details
--  Table Royal_Family: Royal_Family_ID, Royal_Family_Details
--  Table Theme_Parks: Theme_Park_ID, Theme_Park_Details
--  Table Visits: Visit_ID, Tourist_Attraction_ID, Tourist_ID, Visit_Date, Visit_Details
--  Table Photos: Photo_ID, Tourist_Attraction_ID, Name, Description, Filename, Other_Details
--  Table Staff: Staff_ID, Tourist_Attraction_ID, Name, Other_Details
--  Table Tourist_Attraction_Features: Tourist_Attraction_ID, Feature_ID
-- Question: Which transportation method is used the most often to get to tourist attractions?
-- SQL:
SELECT How_to_Get_There FROM Tourist_Attractions GROUP BY How_to_Get_There ORDER BY COUNT(*) DESC
LIMIT 1
```

Table 5.15: The prompt we use for the Spider dataset for few-shot generation with LMs (Part 3), continued from Table. 5.13 and Table. 5.14.

```
-- Translate natural language questions into SQL queries.

-- Example:

-- Database 204_126:
--  Table main_table: id (1), agg (0), place (t1), place_number (1.0), player (larry nelson), country
(united states), score (70-72-73-72=287), score_result (287), score_number (70), score_number1 (70),
score_number2 (72), score_number3 (73), score_number4 (72), to_par (-1), to_par_number (-1.0),
money_lrb_rrb (playoff), money_lrb_rrb_number (58750.0)
-- Question: what was first place 's difference to par ?
-- SQL:
select to_par from main_table where place_number = 1

-- Example:

-- Database 204_522:
--  Table main_table: id (1), agg (0), boat_count (4911), boat_count_number (4911), boat_count_minimum
 (4951), boat_count_maximum (4955), name (ha-201), builder (sasebo naval arsenal), laid_down
(01-03-1945), laid_down_number (1), laid_down_parsed (1945-01-03), laid_down_year (1945),
laid_down_month (1), laid_down_day (3), launched (23-04-1945), launched_number (23), launched_parsed
(1945-04-23), launched_year (1945), launched_month (4), launched_day (23), completed (31-05-1945),
completed_number (31), completed_parsed (1945-05-31), completed_year (1945), completed_month (5),
completed_day (31), fate (decommissioned 30-11-1945. scuttled off goto islands 01-04-1946)
-- Question: when was a boat launched immediately before ha-206 ?
-- SQL:
select name from main_table where launched_parsed < ( select launched_parsed from main_table where
name = 'ha-206' ) order by launched_parsed desc limit 1

-- Example:

-- Database 204_877:
--  Table main_table: id (1), agg (0), place (1), place_number (1.0), position (mf), number (4),
number_number (4.0), name (ryan hall), league_two (10), league_two_number (10.0), fa_cup (1),
fa_cup_number (1.0), league_cup (0), league_cup_number (0.0), fl_trophy (3), fl_trophy_number (3.0),
total (14), total_number (14.0)
-- Question: who scored more , grant or benyon ?
-- SQL:
select name from main_table where name in ( 'anthony grant' , 'elliot benyon' ) order by total_number
desc limit 1

-- Example:

-- Database 204_400:
--  Table main_table: id (1), agg (0), district (1), district_number (1.0), senator (kenneth p.
lavalle), party (republican), caucus (republican), first_elected (1976), first_elected_number (1976),
counties_represented (suffolk), counties_represented_length (1)
--  Table t_counties_represented_list: m_id (1), counties_represented_list (suffolk)
-- Question: how many republicans were elected after 2000 ?
-- SQL:
select count ( * ) from main_table where party = 'republican' and first_elected_number > 2000
```

Table 5.16: The prompt we use for the WTQ dataset for few-shot generation with LMs (Part 1).

```
-- Example:

-- Database 203_208:
--  Table main_table: id (1), agg (0), team (dinamo minsk), location (minsk), venue (dinamo, minsk),
capacity (41040), capacity_number (41040.0), position_in_1993_94 (1), position_in_1993_94_number (1.0)
--  Table t_venue_address: m_id (1), venue_address (dinamo)
-- Question: what is the number of teams located in bobruisk ?
-- SQL:
select count ( team ) from main_table where location = 'bobruisk'

-- Example:

-- Database 203_60:
--  Table main_table: id (1), agg (0), outcome (winner), no (1), no_number (1.0), date (20 july 1981),
 date_number (20), date_parsed (1981-07-20), date_year (1981), date_month (7), date_day (20),
championship (bastad, sweden), surface (clay), opponent_in_the_final (anders jarryd),
score_in_the_final (6-2, 6-3), score_in_the_final_length (2)
--  Table t_championship_address: m_id (1), championship_address (bastad)
--  Table t_score_in_the_final_list: m_id (1), score_in_the_final_list (6-2)
--  Table t_score_in_the_final_list_first: m_id (1), score_in_the_final_list_first (6-2)
--  Table t_score_in_the_final_list_second: m_id (7), score_in_the_final_list_second (1-7)
--  Table t_score_in_the_final_list_first_number: m_id (1), score_in_the_final_list_first_number (6)
--  Table t_score_in_the_final_list_first_number1: m_id (1), score_in_the_final_list_first_number1 (6)
--  Table t_score_in_the_final_list_first_number2: m_id (1), score_in_the_final_list_first_number2 (2)
--  Table t_score_in_the_final_list_second_number: m_id (7), score_in_the_final_list_second_number (1)
--  Table t_score_in_the_final_list_second_number1: m_id (7), score_in_the_final_list_second_number1
(1)
--  Table t_score_in_the_final_list_second_number2: m_id (7), score_in_the_final_list_second_number2
(7)
-- Question: which month were the most championships played ?
-- SQL:
select date_month from main_table group by date_month order by count ( * ) desc limit 1

-- Example:

-- Database 203_462:
--  Table main_table: id (1), agg (0), year (2006), year_number (2006), division (4), division_number
(4.0), league (usl pdl), regular_season (4th, heartland), regular_season_length (2), playoffs (did not
 qualify), open_cup (did not qualify)
--  Table t_regular_season_list: m_id (1), regular_season_list (4th)
-- Question: what year was more successful , 2012 or 2007 ?
-- SQL:
select year_number from main_table where year_number in ( 2012 , 2007 ) order by regular_season limit
1

-- Example:

-- Database 204_139:
-- Question: are their any other airports that are type `` military/public '' besides eagle farm
airport ?
--  Table main_table: id (1), agg (0), community (antil plains), airport_name (antil plains aerodrome)
, type (military), coordinates (19 26'36''s 146 49'29''e/19.44333 s 146.82472 e)
-- SQL:
select ( select count ( airport_name ) from main_table where type = 'military/public' and airport_name
 != 'eagle farm airport' ) > 0
```

Table 5.17: The prompt we use for the WTQ dataset for few-shot generation with LMs (Part 2).

```
## Cristina, John, Clarissa and Sarah want to give their mother a photo album for her birthday.
Cristina brings 7 photos, John brings 10 photos and Sarah brings 9 photos. If the photo album has 40
slots available, how many photos does Clarissa need to bring in order to complete the photo album?
n_photo_cristina = 7
n_photo_john = 10
n_photo_sarah = 9
n_photo_total = n_photo_cristina + n_photo_john + n_photo_sarah
n_slots = 40
n_slots_left = n_slots - n_photo_total
answer = n_slots_left

## Katy, Wendi, and Carrie went to a bread-making party.  Katy brought three 5-pound bags of flour.
Wendi brought twice as much flour as Katy, but Carrie brought 5 pounds less than the amount of flour
Wendi brought.  How much more flour, in ounces, did Carrie bring than Katy?
pound_flour_katy = 3 * 5
pound_flour_wendi = pound_flour_katy * 2
pound_flour_carrie = pound_flour_wendi - 5
pound_diff_carrie_katy = pound_flour_carrie - pound_flour_katy
ounce_diff_carrie_katy = pound_diff_carrie_katy * 16
answer = ounce_diff_carrie_katy

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours.  How many mg does he take a day?
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day

## Kyle bakes 60 cookies and 32 brownies. Kyle eats 2 cookies and 2 brownies. Kyle's mom eats 1 cookie
 and 2 brownies. If Kyle sells a cookie for $1 and a brownie for $1.50, how much money will Kyle make
if he sells all of his baked goods?
n_cookies = 60
n_brownies = 32
n_cookies_left_after_kyle = n_cookies - 2
n_brownies_left_after_kyle = n_brownies - 2
n_cookies_left_after_kyle_mom = n_cookies_left_after_kyle - 1
n_brownies_left_after_kyle_mom = n_brownies_left_after_kyle - 2
money_earned_kyle = n_cookies_left_after_kyle_mom * 1 + n_brownies_left_after_kyle_mom * 1.5
answer = money_earned_kyle
```

Table 5.18: The prompt we use for the GSM8k dataset for few-shot generation with LMs. (Part 1)

```
## There were 63 Easter eggs in the yard.  Hannah found twice as many as Helen.  How many Easter eggs
did Hannah find?
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah

## Ethan is reading a sci-fi book that has 360 pages. He read 40 pages on Saturday morning and another
 10 pages at night. The next day he read twice the total pages as on Saturday. How many pages does he
have left to read?
n_pages = 360
total_page_saturday = 40 + 10
total_page_next_day = total_page_saturday * 2
total_pages_read = total_page_saturday + total_page_next_day
n_pages_left = n_pages - total_pages_read
answer = n_pages_left

## A library has a number of books. 35%
percent_books_for_children = 0.35
percent_books_for_adults = 1.0 - percent_books_for_children
n_books_for_adults = 104
n_books_in_total = n_books_for_adults / percent_books_for_adults
answer = n_books_in_total

## Tyler has 21 CDs. He gives away a third of his CDs to his friend. Then he goes to the music store
and buys 8 brand new CDs. How many CDs does Tyler have now?
n_cds_tyler = 21
percent_cds_given_away =  1.0 / 3.0
n_cds_left_after_giving_away = n_cds_tyler - n_cds_tyler * percent_cds_given_away
n_new_cds_purchased = 8
n_cds_now = n_cds_left_after_giving_away + n_new_cds_purchased
answer = n_cds_now
```

Table 5.19: The prompt we use for the GSM8k dataset for few-shot generation with LMs. (Part 1)

```
# Write Python function to complete the task and pass the assertion tests.

### Task Start ###
# These are the assertions for your function:
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)

""" Write a function to find the similar elements from the given two tuple lists. """
def similar_elements(test_tup1, test_tup2):
    res = tuple(set(test_tup1) & set(test_tup2))
    return (res)
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert is_not_prime(2) == False

""" Write a python function to identify non-prime numbers. """
import math
def is_not_prime(n):
    result = False
    for i in range(2,int(math.sqrt(n)) + 1):
        if n %
            result = True
    return result
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]

""" Write a function to find the largest integers from a given list of numbers using heap queue
algorithm. """
import heapq as hq
def heap_queue_largest(nums,n):
    largest_nums = hq.nlargest(n, nums)
    return largest_nums
### Task End ###
```

Table 5.20: The prompt we use for the MBPP dataset for few-shot generation with LMs.

Chapter 6

Evaluating Language-to-Code Generation Capabilities of LMs

In this chapter, we present `L2CEval`, a unified evaluation framework for language-to-code generation tasks with comprehensive evaluation results for more than 50 models across different sizes, training data mixture and training methods. More specifically, `L2CEval` targets on three representative domains of language-to-code generation, *i.e.,* semantic parsing, math reasoning and Python programming. Through the lens of these tasks, we study the scaling effect of model sizes and training compute; the effect of different training data mixture on model performance; model robustness and calibration measurements, and various other modeling factors such as instruction-tuning and prompting methods. Moreover, given the growing concern of data contamination issues for evaluating LLMs on various generation tasks, including code generation, we develop methods to quantify the data contamination issues for popular code generation benchmarks and present several analysis on the model performance and the similarity to training data for each example at test time. We also release the code for our evaluation framework, as well as model outputs for reproducibility and future studies. [1]

1. The content of this chapter is directly adapted from *"L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models"* by Ni *et. al* , 2024 and *"Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models"* by Riddell *et. al* , 2024.

6.1 Introduction

Language-to-code (L2C[2]) is a type of task that aims to automatically map natural language descriptions to programs, which are later executed to satisfy the user's demand (Austin et al., 2021a; Yin and Neubig, 2017). As illustrated in Figure. 6.1, language-to-code is the foundation of many applications in AI, such as task-oriented dialogue systems (Andreas et al., 2020), coding assistant (Agashe et al., 2019b; Lai et al., 2022), language interfaces to databases (Pasupat and Liang, 2015; Yu et al., 2018b), and robotic control (Shridhar et al., 2020; Zhou et al., 2021). It has also served as a great testbed for evaluating various language understanding capabilities of NLP systems, such as logical and math reasoning (Gao et al., 2022; Han et al., 2022), grounded language understanding (Huang et al., 2022; Xie et al., 2022), and tool use (Paranjape et al., 2023; Schick et al., 2023).

Recent progress on large language models (LLMs) (Chowdhery et al., 2022; OpenAI, 2023; Touvron et al., 2023a), especially those specifically trained for coding (Chen et al., 2021a; Fried et al., 2022; Li et al., 2023; Nijkamp et al., 2022), has shown that LLMs trained on a mixture of text and code are able to perform language-to-code generation under few-shot or even zero-shot learning settings (Ni et al., 2023a; Rajkumar et al., 2022). However, the modeling factors that affect the performance of LLMs for such L2C tasks—including model size, training data mixture, prompting methods, and instruction tuning—are poorly understood. In addition, there lacks a consistent evaluation of different LLMs on the same spectrum of language-to-code tasks, making it difficult for the users to decide which models to use for certain tasks or if they should resort to finetuning their own models. Beyond model performance, model properties such as robustness to prompt and confidence calibration are also crucial for understanding the reliability of LLMs, but such properties have not been systematically studied for L2C tasks.

In this work, we present L2CEval, providing a systematic evaluation of the language-to-code generation capabilities of LLMs. L2CEval includes a wide range of state-of-the-art models, specifically 56 models from 13 different organizations, all evaluated on three core domains of language-to-code generation tasks: *semantic parsing*, *math reasoning*, and *Python*

2. We refer to "natural language" whenever we use the term "language" in this work.

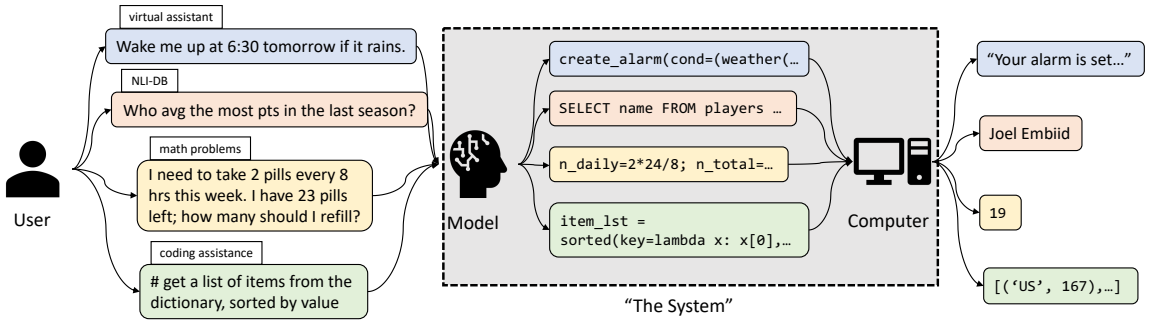


Figure 6.1: Language-to-code (L2C) generation is the cornerstone for many applications in AI. It is also the key to enabling direct communication between the users and the computers with natural language.

programming. Our L2CEval framework includes extensive evaluations of models as small as 1 billion parameters, to significantly larger ones such as Falcon-180B, as well as davinci and GPT-4 models from OpenAI. We also benchmark models that are trained on different mixtures of data of varying sizes (35B $\sim$ 3.5T tokens), as well as models that are instruction-tuned, from both open-source and open-access proprietary categories. Our work is the first to conduct extensive and thorough comparisons of LLMs for language-to-code generation across multiple dimensions of variation. To summarize, we release L2CEval and its main contributions are as follows:

- We standardize the evaluation (*e.g.,* prompts, metrics) of **7** L2C tasks across domains of semantic parsing, math reasoning, and Python programming to allow controlled comparisons among **56** models from **13** organizations;
- We study the scaling effect of model size, pretraining compute/data mixture, as well as several modeling contributions (*e.g.,* instruction-tuning, zero/few-shot prompting) for L2C tasks;
- We analyze the robustness and calibration measurements of different models, and identify their common error cases;
- We release the code for our evaluation framework, and model outputs (*i.e.,* texts and logits) for reproducibility and future studies.

Through our work, we hope to provide insight into applying LLMs to L2C applications, as well as building future LLMs.

Domain: Datasets	Split	Size	Input	Output
Semantic Parsing:				
Spider (Yu et al., 2018b)	Dev	1,032	DB schema + NL	SQL Query
WikiTQ (Pasupat and Liang, 2015)	Dev	2,831	Table headers* + NL	SQL Query
Math Reasoning:				
GSM8k (Cobbe et al., 2021)	Dev[3]	1,495	Math problem in NL	Python solutions
SVAMP (Patel et al., 2021)	All	1,992	Math problem in NL	Python solutions
Python Programming:				
MBPP (Austin et al., 2021a)	Test	500	NL spec. + 1 test	Python functions
HumanEval (Chen et al., 2021a)	All	164	NL spec. + 1-3 test	Python functions
DS-1000 (Lai et al., 2022)	All	1,000	NL spec.	Python lines

Table 6.1: A summary of all the benchmarks included for evaluation in `L2CEval`.

6.2 L2CEval

The main motivation behind `L2CEval` is to provide a *comprehensive* evaluation of language-to-code generation capabilities and understand what affects such `L2C` capabilities. In the following sections, we first discuss the key desiderata in design of `L2CEval` in § 6.2.1, then the formulation of `L2C` in § 6.2.2, then in § 6.2.3 we introduce the domains and specific tasks we consider for `L2CEval`, as well as the reasons for choosing such tasks. Finally, in § 6.2.4, we describe the models included in `L2CEval` and the model selection process.

6.2.1 Desiderata

To ensure that the `L2CEval` framework serves as a comprehensive resource for evaluating language-to-code (`L2C`) capabilities, we outline a set of key desiderata that guided its construction.

Task inclusion. *Diverse Task Representation*: The benchmark aims to capture a wide scope of L2C tasks, specifically incorporating semantic parsing, Python programming, and math reasoning. *Task complexity*: Under each of these domains, we include 2 to 3 sub-tasks to represent a combination of different levels of language understanding, reasoning, and programming abilities.

Model Evaluation. *Open Source and Commercial Models*: `L2CEval` is designed to accommodate both open-source and commercial models to provide a holistic view of available L2C capabilities. Our focus is more on the open-source models, however, as proprietary models do *not* disclose certain basic information which makes drawing scientific conclusions

difficult. *Model Size Variability*: The benchmark includes models of different sizes to explore any correlation between model size and performance. *Specialized vs General Models*: We examine the performance trade-offs between models exclusively trained on code and general language models to understand the advantages or disadvantages of specialization.

Evaluation setup. *Standardized Prompts*: All tasks and models are evaluated using standardized prompts, overcoming the inconsistencies prevalent in prior work. *Reproducibility*: Our evaluation setup is clearly described to facilitate reproducible experiments. *Fair Comparison*: Universal evaluation metrics are employed across diverse tasks and models to enable equitable comparisons.

Transparency and reusability. *Documentation*: The framework is thoroughly documented to promote community engagement and constructive feedback. *Interoperability*: `L2CEval` is built to be easily updated or extended, allowing for the incorporation of new tasks or models as the field evolves. We also share the code and model outputs to support future research in this domain.

By adhering to these desiderata, the `L2CEval` framework aims to be a comprehensive, fair, and practical evaluation resource for the community.

6.2.2 Language-to-Code Generation (`L2C`)

Problem Formulation. While language-to-code generation covers a wide range of tasks as shown in Figure. 6.1, here we attempt to give a unified problem formulation. Given the user's intent described in natural language x (*e.g.,* description of a Python function) and optionally some programming context c (*e.g.,* existing function definitions, open test cases), an `L2C` model aims to automatically map the input to a program y (*e.g.,* a Python function). In the context of LLMs, this is typically modeled as a conditional generation problem with prompting:

$$\hat{y} = \arg\max_{y} P_{\mathbf{LM}}(y \mid \mathbf{prompt}) \tag{6.1}$$

3. Here we use the split from Ni et al. (2022).

where the "**prompt**" consists of task-specific instructions I and optionally m exemplars $\{(x_i, y_i, c_i)\}_{i<m}$:

$$\textbf{prompt} = f(I, \{(x_i, y_i, c_i)\}_{i<m}, c, x)$$

To obtain the best candidate program $\hat{y}$, we use *greedy decoding* in all our evaluations.[4] Also for a fair comparison, we standardize the prompting methods by following previous work (Ben Allal et al., 2022; Ni et al., 2023a) and avoid prompts that are tailored for specific models.

Execution-based Evaluation. The generated program candidate $\hat{y}$, sometimes accompanied with additional execution context e (*e.g.,* connection to DB) is later executed by an executor $\mathcal{E}(\cdot)$ (*e.g.,* Python interpreter). We can evaluate *execution accuracy* by checking if it matches the gold execution results z^* upon execution:

$$\text{Acc.} = \mathbb{1}(\hat{z}, z^*) \text{ where } \hat{z} = \mathcal{E}(\hat{y}, e) \tag{6.2}$$

We use execution accuracy as a proxy for whether the user's original intent is satisfied.[5] This is also consistent with previous work on `L2C` (Ni et al., 2023a; Shi et al., 2022a; Xie et al., 2022).

6.2.3 Tasks

We evaluate the language-to-code capabilities of LLMs in three representative application scenarios shown in Figure. 6.1: *semantic parsing*, *math reasoning*, and *Python programming*. Particularly, these tasks collectively assess the capabilities of models in language-to-code generation to understand natural language in different contexts, reason about the steps for solving the problem, and convert it into executable code (see Figure. 6.1). *Semantic parsing* focuses on the transformation of natural language queries into structured, domain-specific languages; *math reasoning* challenges the models' numerical and logical reasoning abilities by requiring them to solve problems that involve multiple steps of calculation and reasoning; and

4. We discuss the limitation of greedy decoding in § 6.7.

5. See § 6.7 for the limitations of execution-based evaluation.

Organization	Model Series	Variants	Sizes	# All Tokens	# Code Tokens	Context Length	Code Specific
Salesforce	CodeGen (Nijkamp et al., 2022)	multi/mono	6.1/16.1B	505∼577B	119∼191B	2K	✓
	CodeGen-2.5 (Nijkamp et al., 2023)	multi/mono/ instruct†	7B	1.4T	1.4T	2K	✓
Eleuther AI	GPT-J (Wang and Komatsuzaki, 2021)		6.1B	402B	46B	2K	✗
	GPT-NeoX (Black et al., 2022)		20.6B	472B	54B	2K	✗
	Pythia (Biderman et al., 2023a)		1.4/6.9/12B	300B	35B	2K	✗
Databricks	Dolly-v2 (Conover et al., 2023)		6.9/12B	–	–	2K	✗
BigCode	SantaCoder (Allal et al., 2023)		1.1B	236B	236B	2K	✓
	StarCoder (Li et al., 2023)	base/plus	15.5B	1∼1.6T	1T	8K	✓
Meta	InCoder (Fried et al., 2022)		1.3/6.7B	52B	52B	2K	✓
	LLaMA (Touvron et al., 2023a)		7/13/30B	1∼1.4T	45∼63B	2K	✗
	LLaMA-2 (Touvron et al., 2023b)		7/13/70B	2T	-	4K	✗
	CodeLLaMA (Roziere et al., 2023)	base/instruct†	7/13/34B	2.5T	435B	16K	✓
Stanford	Alpaca† (Taori et al., 2023)		7/13/30B	–	–	2K	✗
Replit	Replit-v1-3b (rep)		3B	525B	525B	2K	✓
WizardLM	WizardCoder-v1 (Luo et al., 2023)		15B	–	–	2K	✓
MosaicML	MPT (Team, 2023a,b)	base/instruct†	7/30B	1T	135B	2K/8K	✗
MistralAI	Mistral-v0.1 (Jiang et al., 2023a)	base/instruct†	7B	–	–	32K	✗
XLANG	Lemur-v1 (Xu et al., 2023)		70B	–	–	4K	✓
TII	Falcon (Almazrouei et al., 2023)	base/instruct†	7/40/180B	1∼3.5T	–	2K	✗
OpenAI	Codex (Chen et al., 2021a)	code-cushman-001 code-davinci-002	12B/ –	400B	100B	2K/8K	✓
	InstructGPT† (Ouyang et al., 2022)	text-davinci-002/3	–	–	–	4K	✗
	ChatGPT† (OpenAI, 2022)	turbo-0301/0613	–	–	–		
	GPT-4† (OpenAI, 2023)	0314/0613	–	–	–	8K	✗

Table 6.2: Information table for the models evaluated in this work. –: no information on training data size is available, or the model is further tuned on top of other models. †: Instruction-tuned models.

Python programming tests the models' proficiency in generating functional code that aligns with a user's intent, reflecting a real-world application of LLMs in software development. A summary of `L2CEval` benchmarks is shown as Table. 6.1 and we discuss each of these tasks in detail as below.

Semantic parsing. Semantic parsing considers the task of translating a user's natural language utterance (*e.g., who averaged the most pots in the last season?* in Figure. 6.1) into machine-executable programs (*e.g.,* an SQL database query), and has been a long-standing problem in NLP (Berant et al., 2013; Zettlemoyer and Collins, 2005). A prompt to an LLM consists of an NL utterance and descriptions of relevant structured context, such as the schema information of a database (*e.g.,* columns in each table). The target output is a program defined in some domain-specific languages, such as SQL. Intuitively, semantic parsing challenges LLMs on grounded language understanding (Cheng et al., 2022; Xie et al., 2022), where a model needs to associate NL concepts in utterances (*e.g., "last season"*) with relevant structured knowledge (*e.g.,* superlative operation on column `season`) in order to synthesize the program (Pasupat and Liang, 2015; Yin et al., 2020; Yu et al., 2018b). In this

work, we choose to use text-to-SQL as a representative task as it closely ties with applications such as natural language interface to databases (Affolter et al., 2019; Androutsopoulos et al., 1995). Recent work (Ni et al., 2023a; Rajkumar et al., 2022) shows that LLMs are effective in performing text-to-SQL parsing. In this work, we use two widely-used text-to-SQL datasets, **Spider** (Yu et al., 2018b) and **WikiTQ** (Pasupat and Liang, 2015), as our datasets for benchmarking semantic parsing capabilities of LLMs. Following Xie et al. (2022), we concatenate the natural language utterance with the database schema or table headers as LLM input.[6]

Math reasoning. To solve a math word problem, a model needs to abstract the mathematical relations from the natural language description, and reason about the potential steps. Compared to semantic parsing where the target programs are table-lookup queries, programs for math reasoning tasks usually require multiple steps of calculation and numerical and logical reasoning. Because of this, math word problems are widely adopted as testbeds for evaluating the reasoning abilities of LLMs (Cobbe et al., 2021; Ni et al., 2022; Wei et al., 2022c; Welleck et al., 2022). In this chapter, we choose the **GSM8k** (Cobbe et al., 2021) and **SVAMP** (Patel et al., 2021) datasets, which are grade-school level math problems described in natural language. We chose these two benchmarks due to their moderate difficulty and popularity. Following Welleck et al. (2022) and Gao et al. (2022), we prompt the models to answer math word problems by generating Python programs as solutions, which are later executed by a Python interpreter to output the answer.

Python programming. One of the most important applications for LLMs trained on code is to assist programmers in developing software. Typically, a model is given a developer's natural language intent (*e.g., write a merge sort function*) with optional additional specifications (Austin et al., 2021a) such as input/output examples or unit tests (*e.g.,* `assert merge_sort([5,7,3])==[3,5,7]`), to generate the code (*e.g.,* a Python function) that implements the user's intent. To evaluate the basic programming skills of the LLMs, we

6. While more challenging datasets exists for text-to-SQL (*e.g.,* BIRD-SQL (Li et al., 2024)), we believe the observations should be generalizable due to the similar task format.

use the **MBPP** (Austin et al., 2021a) and **HumanEval** (Chen et al., 2021a) datasets, for which the model needs to implement some basic Python functions to pass the test cases. Moreover, we also include **DS-1000** (Lai et al., 2022), which focuses on data-science-related questions and libraries. [7]

More task-specific settings are described in § 6.5.2, and example input outputs for different tasks are shown in § 6.5.1.

6.2.4 Models

We evaluate 56 models that vary in size, training data mixture, context length, and training methods. Table. 6.2 summarizes the open-source models we evaluated and several key properties.

Selection criteria. While it is not possible to evaluate every single LLM on these tasks, we strive to provide a comprehensive evaluation of the current LLMs in `L2C` generation, by covering a diversified selection of LLMs of varying sizes and are trained on different mixtures of data. For example, the size of the models we consider ranges from 1B (*e.g.,* SantaCoder (Allal et al., 2023)) to 170B+ (*e.g.,* Falcon-180B (Almazrouei et al., 2023) and GPT-4 model from OpenAI). Though we prioritize the evaluation of code-specific models, which means that the majority of the training tokens are from code (*e.g.,* CodeLLaMA (Roziere et al., 2023), StarCoder (Li et al., 2023)), we also include the most competitive general LLMs such as LLaMA2-70B (Touvron et al., 2023a) and Falcon-180B for comparison. To evaluate the effect of instruction-tuning and its data mixtures on `L2C` tasks, we also include several instruct-tuned versions of the LLMs, such as Alpaca (Taori et al., 2023), Dolly (Conover et al., 2023), etc. We also prioritize the evaluation of open-source models and mainly present our findings with these models as we are unclear about the technical details of proprietary models (*e.g.,* model size, training data) and hesitate to speculate about them.

7. While APPS (Hendrycks et al., 2021) is another popular Python programming dataset, we decide not to use it due to various issues found in Li et al. (2022b).

Group	Model	Spider (2-shot)	WikiTQ (2-shot)	GSM8k (8-shot)	SVAMP (4-shot)	MBPP (3-shot)	HE (0-shot)	DS-1K (0-shot)	MWR
Other	gpt-4	79.2	56.7	88.5	92.8	74.2	80.5	24.0	100%
	text-davinci-003	68.3	45.4	64.1	80.7	63.6	52.4	15.3	95%
	gpt-3.5-turbo	72.7	38.4	74.7	80.7	66.6	39.0	11.0	92%
20 ∼ 100B	*CodeLLaMA-base (34B)*	61.7	32.3	43.6	70.7	45.6	44.5	22.4	90%
	Lemur (70B)	68.0	44.9	57.5	47.9	51.4	41.5	20.7	88%
	LLaMA-2 (70B)	58.5	37.3	56.0	73.9	36.8	28.7	16.9	84%
10 ∼20B	*CodeLLaMA (13B)*	58.5	35.6	30.7	64.9	44.0	34.2	18.8	86%
	StarCoder (15.5B)	52.1	27.4	22.1	48.8	46.6	34.2	19.8	78%
	LLaMA-2 (13B)	35.7	24.6	26.1	58.9	27.0	17.7	9.1	59%
2 ∼10B	Mistral-v0.1 (7B)	53.3	31.4	38.4	69.4	37.8	25.0	14.1	79%
	CodeLLaMA-base (7B)	54.3	29.5	25.5	52.8	40.0	31.1	16.0	76%
	CodeGen2.5-multi (7B)	53.8	29.6	14.9	43.1	38.2	31.1	16.9	71%
<2B	*SantaCoder (1.3B)*	19.0	11.4	2.8	0.0	26.2	17.7	1.1	24%
	InCoder (1.1B)	13.4	6.2	1.0	3.5	13.8	8.5	2.9	11%
	Pythia (1.4B)	5.7	4.4	1.5	9.3	5.8	3.7	1.8	6%

Table 6.3: Top-3 models at different size ranges. All models are of the "base" variant (*i.e.,* w/o instruction-tuning or RLHF), except the "Other" group, which is for reference purposes only. MWR: Mean Win Rate (see definition in § 6.3.1). The best performance for each group is highlighted with color shades indicating the relative performance across different groups. Code-specific LLMs are noted in *italics*.

Model Access. For all the open-source models, we access them through huggingface model hub[8] and run them locally on a server with RTX A6000 48GB GPUs, using `Lightning`[9] as underlying framework. For proprietary OpenAI models we access them through the public API.[10]

6.3 Results and Analysis

We organize the experiment results and analysis as follows. We first discuss different scaling effects in § 6.3.1, then in § 6.3.2 and § 6.3.3, we analyze how pretraining data mixture and instruction-tuning affects the models for `L2C` tasks. To study model robustness, in § 6.3.4, we measure the sensitivity of the models on the few-shot demonstrations, and show model confidence calibration in § 6.3.5. Finally, we present an error analysis in § 6.3.6.

8. `https://huggingface.co/models`

9. `https://lightning.ai/`

10. `https://api.openai.com/`

6.3.1 Scaling

We examine the correlation between model performance and its parameter count, as well as the pretraining compute. While most of our findings align with previous work on scaling laws (Hoffmann et al., 2022; Kaplan et al., 2020), we focus on properties that are more related to L2C tasks.

Model size. We present the top-3 models across different size ranges based on Mean Win Rate (MWR) in Table. 6.3. MWR is defined as the fraction of times a model outperforms other models, averaged across all 7 tasks. More specifically, given a set of N models for comparison $\mathcal{P} = \{P_1, ..., P_N\}$ and and a set of M tasks $\mathcal{D} = \{\mathcal{D}_1, ..., \mathcal{D}_M\}$, MWR of a model P_i is defined as:

$$\text{MWR}(P_i) = \sum_{k \leq M} \frac{|\{P_j | \mathcal{D}_k(P_j) \leq \mathcal{D}_k(P_i), j \leq N\}|}{M \times N}$$

where $\mathcal{D}_k(P_j)$ denotes the performance of model P_j on task $\mathcal{D}_k$. From this table, we can observe clear performance gaps between models of different size groups. However, such a scaling effect also varies across domains: for math reasoning tasks, the performance gaps between models of different size groups are much larger than those of Python programming and semantic parsing tasks. Given the programs to solve these math problems are relatively simple in syntax, we hypothesize that the bottleneck lies in the planning and reasoning capabilities, which correlate with model size. This hypothesis is consistent with previous findings (Wei et al., 2022a,c). This can be better observed from § 6.6.4 as we plot the scaling curve independently for each task.

Pretraining compute. To study the scaling of compute resources during pretraining, we plot the average model performance across all 7 tasks against the estimated FLOPS of compute[11] needed during training, as shown in Figure. 6.2. From the figure, we can see that code LMs are much more compute efficient than general LMs in terms of L2C performances. This is particularly noticeable when the compute budget is constrained; for example, SantaCoder-1B outperforms Pythia-6.9B with an order of magnitude less compute,

11. Here we base our estimation on (Kaplan et al., 2020): FLOPS $\approx$ 6 * model size (B) * training tokens (B).

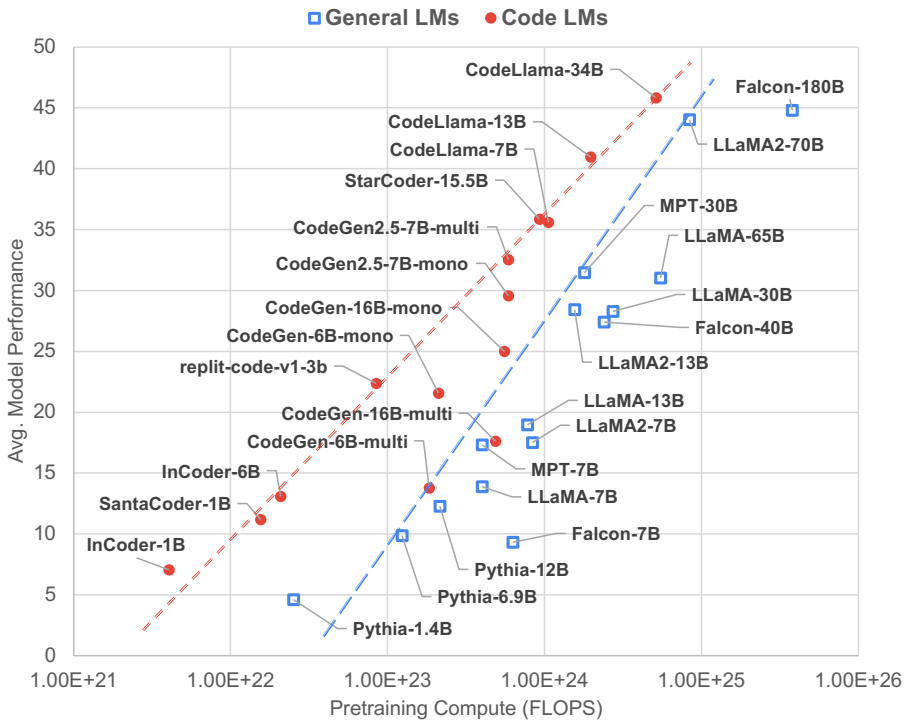


Figure 6.2: Pretraining compute scaling for code-specific and general LMs. Dashed lines denote the trend line where the optimal compute is achieved for models of each category.

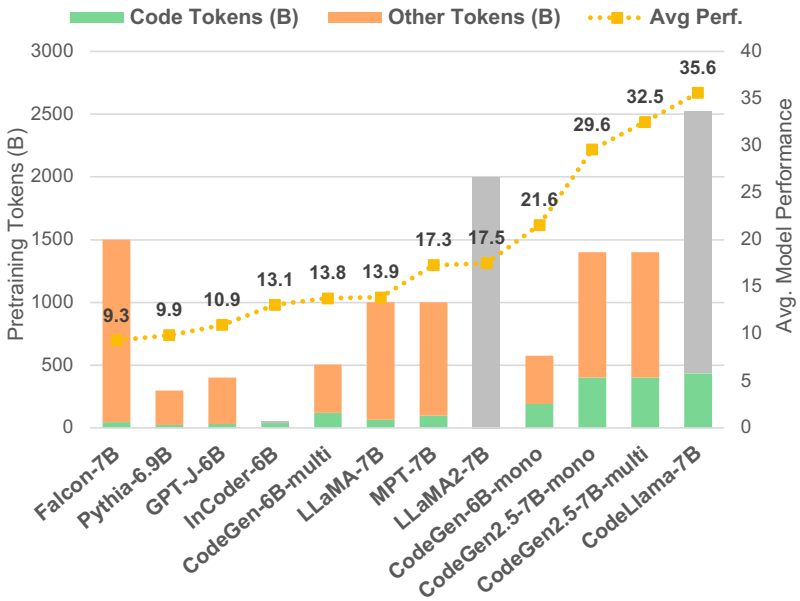


Figure 6.3: Pretraining data mixture for models of similar sizes (6 $\sim$ 7B), ranked by performance. LLaMA-2 paper (Touvron et al., 2023b) only shares the size but not the distribution of the pretraining data, and CodeLLaMA is trained on top of LLaMA-2.

and InCoder-1B outperforms Pythia-1.4B using only 1/5 of the pretraining compute. While

Models	Few-Shot						Zero-Shot					
	Spider (2)		GSM8k (8)		MBPP (3)		Spider (0)		GSM8k (0)		MBPP (0)	
	Base	IT	Base	IT	Base	IT	Base	IT	Base	IT	Base	IT
Pythia/Dolly-6.9B	12.5	*13.1*	2.6	2.6	13.2	12.0	3.0	*5.2*	0.0	0.0	0.4	*9.4*
Pythia/Dolly-12B	16.2	13.0	2.6	2.6	19.0	15.0	2.8	*6.5*	0.0	0.0	1.2	*3.8*
MPT-7B	27.3	25.5	10.9	10.4	21.0	*24.0*	19.5	*27.0*	1.5	*6.2*	0.2	*10.2*
MPT-30B	43.3	42.8	30.7	29.1	29.2	28.4	34.9	*44.1*	0.0	*1.9*	0.8	*23.4*
LLaMA/Alpaca-7B‡	13.1	*16.1*	8.0	3.5	16.6	14.4	5.7	*20.5*	0.0	0.0	5.0	*13.2*
LLaMA/Alpaca-13B‡	15.2	*24.3*	15.7	*18.5*	22.8	*23.4*	15.2	*26.1*	0.0	0.0	2.2	*7.2*
LLaMA/Alpaca-30B‡	38.5	*46.2*	15.9	*19.4*	26.6	*32.0*	41.8	*46.3*	0.0	0.0	20.6	*27.2*
CodeLlama-7B‡	54.3	*55.9*	25.5	*26.5*	40.0	*41.2*	51.8	*56.1*	0.0	*7.2*	10.0	*15.2*
CodeLlama-13B‡	58.5	*63.0*	30.7	*48.2*	44.0	*48.2*	64.2	*66.2*	17.0	10.6	18.6	*19.0*
CodeLlama-34B‡	61.7	*68.7*	43.6	*52.8*	45.6	*52.8*	69.6	*69.7*	15.1	5.8	34.0	*42.8*

Table 6.4: How instruction-tuning affects few/zero-shot L2C performances. Model names shown as {base}/{IT}-{size}. ‡: instruction-tuning includes code-related tasks. "IT" denotes the instruction-tuned version of the base model. Performance *improvements* and degradations are marked accordingly.

code LMs generally require fewer FLOPS to achieve comparable performance on L2C tasks,[12] it is expected as well, given that general LMs are also optimized for many other natural language tasks unrelated to coding. Moreover, such a trend also seems to be diminishing when scaling up (*e.g.,* comparing CodeLLaMA-34B and LLaMA-2-70B), which suggests that when model and pretraining data size gets larger, it is possible that general LMs will be as compute efficient as code LMs for L2C tasks.

6.3.2 Data Mixture

While all of the models we evaluate in L2CEval have seen code during pretraining, the distributions of their training data mixture vary significantly, as illustrated in Table. 6.2. In Figure. 6.3, we show 12 different models that are around 7B, ordered by their average model performance on all 7 tasks, and plot the amount of the code and non-code tokens in their pretraining data. As we can see from the figure, the number of code tokens in the pretraining data affects the model performance much more than the amount of non-code tokens, and the model performance is almost monotonically increasing with the amount of code tokens in the training data. Given that CodeLLaMA models are further pretrained on code tokens on top of LLaMA-2, by comparing the performance of their 7B and 13B

12. The only exceptions are CodeGen-multi/mono models, which are trained with far less amount of code tokens compared with other code LLMs.

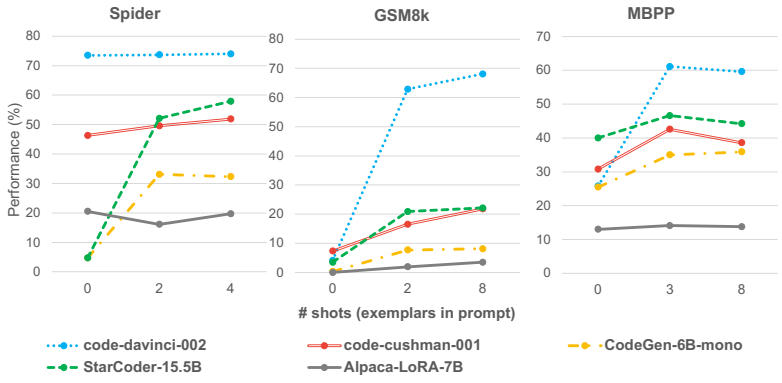


Figure 6.4: Models perf. with different # of exemplars in the prompt.

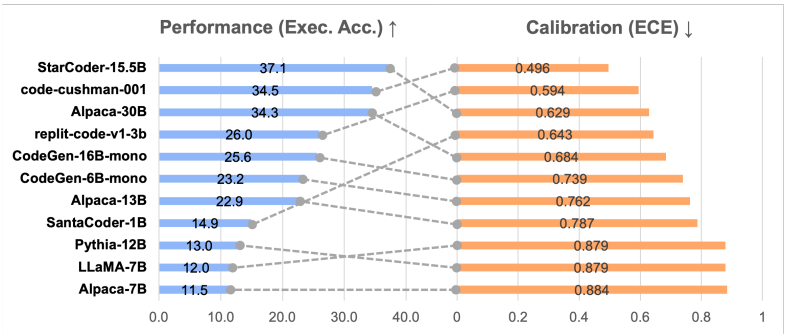


Figure 6.5: Avg. perf. across selected datasets (*i.e.,* Spider, WikiTQ, GSM8k and MBPP) and their calibration score rankings.

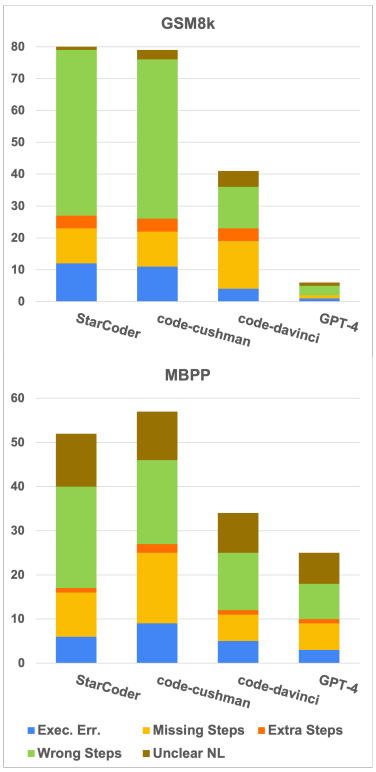


Figure 6.6: Error analysis on 100 examples for GSM8k and MBPP.

versions in Figure. 6.3 and Table. 6.3, we can see that training on more code tokens not only drastically improves text-to-sql parsing and Python programming, but also math reasoning tasks. As mentioned in § 6.3.1, since the generated Python solutions for GSM8K and SVAMP are both simple in syntax (*i.e.,* straight-line programs with numeric operations), we hypothesize that training on more code tokens improves the reasoning abilities of LMs in general. This is also consistent with previous findings (Fu et al., 2022; Mishra et al., 2022; Wei et al., 2022c).

6.3.3 Instruction-tuning

Instruction tuning (Ouyang et al., 2022) is a type of method that enhances the ability of LLMs to follow instructions written in natural language. Here we compare the instruction-tuned models and their base models and show their few/zero-shot results in Table. 6.4.

Zero-shot results. Firstly, we observe that the zero-shot results are generally improved after instruction tuning for all models on Spider and MBPP. This is perhaps a little surprising for Dolly and MPT-instruct models as their instruction-tuning data does not explicitly include coding tasks. Besides the fact that instruction-tuning trains the models to focus more on the instruction, we also hypothesize that instruction-tuning generally improves language understanding abilities, which is essential for `L2C` tasks. We also note that the zeros-shot performances for GSM8k are all zeros for the selected models. By inspecting the model outputs, we find that the models fail to follow the instructions and provide the answer by ending the Python solution with `answer = x`.

Few-shot results. As for few-shot performance numbers, those that are instruction-tuned with coding tasks, such as Alpaca and CodeLLaMA-instruct models, yield much more consistent improvements than Dolly and MPT-instruct across all tasks. Notably, CodeLLaMA-34B-instruct improves 7.0, 9.2, and 7.2 points over the base model on Spider, GSM8K, and MBPP datasets, respectively. Though we observe that some few-shot results deteriorate for Dolly and MPT-instruct, it should also be noted that such performance degradations are quite minimal, as half of them are within 2%. It is suggested in (Ouyang et al., 2022) that instruction-tuning generally decreases few-shot performance, as it shifts the attention of the model from the few-shot exemplars to the instructions, but from these results, we believe that whether instruction-tuning improves few-shot performance largely depends on how similar the instruction-tuning tasks are to tasks for evaluation.

6.3.4 Sensitivity to Prompt

Here we study how sensitive are the models to the number of few-shot demonstrations or different examples in the prompt.

Number of few-shot demonstrations. Figure. 6.4 illustrates the correlation between model performance and the number of exemplars in the prompt[13]. While increasing the

13. While the range of number of shots are different for each task due to different task prompt lengths (*e.g.,* database schema encoding for Spider), we keep it consistent across different models on the same task for a fair comparison.

number of few-shot exemplars in the prompt generally improves execution accuracy, such improvement is not consistent with different models and tasks. For example, on the MBPP dataset, increasing from 3 to 8 exemplars in the prompt actually decreases the performance for most of the selected models, *e.g.,* by 4.0% for codex-cushman. We hypothesize that this is because the programs in the prompt will bias the model towards generating similar programs and ignore the specification. A supporting evidence of this is in Table. 6.5, as codex-cushman is shown to be more sensitive to the exemplars. This effect has also been observed in Li et al. (2022b).

Different examples as demonstrations. Moreover, we also show the sensitivity of the models to different exemplars and present the results in Table. 6.5 by showing the variance of model performance across different runs using different exemplars in the prompt. While the variances differ for different models and tasks, none of them are significant enough to alter the ranking of the models, nor impact the conclusions presented in this work.

6.3.5 Model Calibration

A good model not only produces high-quality outputs, but also should be well-calibrated, meaning that it should be uncertain about its predictions when such predictions are wrong. Following recent work (Liang et al., 2022), we evaluate model calibration using *expected calibration error* (ECE) (Guo et al., 2017; Naeini et al., 2015). For a model $P_{\mathbf{LM}}$ and a dataset $\mathcal{D}$, this is defined as:

$$\begin{aligned}\text{ECE}(P_{\mathbf{LM}}, \mathcal{D}) =& \mathbb{E}_{(x,y^*)\sim\mathcal{D}}[P_{\mathbf{LM}}(\hat{y}|x) - \text{Acc}(\hat{y}, y^*)] \\ &\text{s.t. } \hat{y} = \arg\max_y P_{\mathbf{LM}}(y|x)\end{aligned}$$

where $\text{Acc}(\cdot)$ denotes the execution accuracy metric described as Equation. 6.2. From the results shown in Figure. 6.5, we can observe that while model calibration generally correlates with model performance, the best-performing models are not necessarily the ones with the best calibration. Note that with a well-calibrated model, methods such as voting (Li et al., 2022a; Wang et al., 2022b) and confidence-based reranking (Ni et al., 2023a) may be used

Models	Spider (2-shot)	GSM8k (2-shot)	MBPP (3-shot)
code-davinci	73.7±0.3	66.4±1.0	59.0±1.9
code-cushman	50.4±0.7	24.2±1.1	39.3±3.3
CodeGen-6B-mono	32.4±0.6	13.8±0.2	35.5±0.5
StarCoder-15.5B	54.9±2.7	32.3±0.8	44.1±2.2
Alpaca-7B	20.1±3.5	7.3±1.2	13.6±0.6

Table 6.5: Mean and std for (n)-shot performance over 3 runs with different random exemplars.

to further improve their performance. Moreover, a better-calibrated model is more reliable in practical applications, such as coding assistants, where its confidence levels can serve as indicators of generation quality.

6.3.6 Error Modes

In Figure. 6.6, we present an error analysis on the four best models, by manually[14] examining a fixed set of 100 examples from the GSM8k and MBPP datasets across 4 selected models. We categorize the errors into 5 cases:

1) *execution error*, where deformed programs are generated;

2/3) *missing/extra steps*, where some key steps are missing or extraneous lines are generated in predicted code;

4) *wrong steps*, where the model only makes subtle mistakes in certain steps in the code;

5) when the NL specification itself is ambiguous and *unclear*.

From the results shown in Figure. 6.6, we can see that for GSM8k, compared with stronger models (*e.g.,* code-davinci and GPT-4), while a similar number of errors are made for missing and generating extra steps for solving the math problem, StarCoder and code-cushman make more mistakes in predicting intermediate steps, or generating deformed programs. On MBPP however, weaker models are prone to miss crucial steps in the implementation, which shows a lack of understanding of the problem as well as planning abilities. Hallucination (Ji et al., 2023) is a common issue in natural language generation, while we find it to be rare for models to generate lines of code that are extraneous, hallucination can also exhibit as using

14. Two of the authors performed this annotation.

wrong operators or introducing variable values that do not exist in the natural language description, which would be categorized as "wrong steps" in Figure. 6.6.

6.4 Data Contamination

To quantify data contamination for code LLMs, we first introduce methods used to measure program similarity from surface- and semantic-level and obtain the final similarity score in § 6.4.1. Next, in § 6.4.2 we introduce the experiment setup and later in § 6.4.3 we show the main results.

6.4.1 Measuring Program Similarity

While most popular code generation benchmarks focus on generating functions, the training data are often chunked by files, which may contain multiple functions or classes. This means that document-level de-duplication techniques (Allamanis, 2019) cannot be used effectively, as other programs within the document may add too much noise. Thus we opt to perform substring-level matching, which is much more computationally heavy but also more accurate than methods used by previous work (Lee et al., 2022; Peng et al., 2023). More specifically, we use a sliding window to scan the training data character-by-character and compute its similarity scores with gold solutions in the benchmarks. To maximize the recall of possible contaminated examples in coding benchmarks, we employ both surface- and semantic-level similarity measurements.

Surface-Level Similarity. To measure surface-level similarity between programs, we use the *Levenshtein similarity score* (Sarkar et al., 2016), which is the Levenshtein edit distance (Levenshtein, 1965) normalized by the length of both the source and target strings. We selected the Levenshtein similarity score as the first step in our pipeline because it is an easy-to-compute and intuitive measurement that can handle surface-level fuzzy matches between two programs. While the Levenshtein edit distance has been used before to deduplicate datasets at a file level (Chowdhery et al., 2022), we perform it on a substring level. [15]

15. we use the `rapidfuzz` python library to calculate the similarity score `https://pypi.org/project/rapidfuzz/`.

Semantic Similarity. While the surface-level similarity metrics can easily capture similar programs in surface form, two semantically similar or even identical programs can have very different surface form due to different identifiers (*e.g.,* variable names) or whitespace characters. Therefore, finding semantically similar programs is also crucial for measuring contamination and understanding the generalization capabilities of the models. To measure semantic similarity between programs, we adopt the *Dolos toolkit* (Maertens et al., 2022), which is a source code plagiarism detection tool for education purposes. Dolos first uses `tree-sitter`[16] to tokenize and canonicalize the program into representations of abstract syntax trees (ASTs), then computes a similarity score representing the semantic-level similarity based on the k-gram matching between source and target programs. Since Dolos measures similarities based on the ASTs, non-semantic changes that greatly decrease the Levenshtein similarity scores, such as indentations and variable/function names, will not affect the scores calculated by Dolos. Dolos was also used in previous works for detecting intellectual property violations (Yu et al., 2023).

Aggregating Similarity Scores. We use a two-stage process to analyze test examples and their correct (gold standard) program solutions. First, we measure the surface-level similarity by calculating Levenshtein scores. This involves comparing all substrings of the same length as the gold solution across all relevant files in specific subsets of our dataset (see § 6.4.2 for details). We keep the top 500 programs[17] with the highest Levenshtein similarity scores for each test example for the next step. With the top 500 programs with the highest Levenshtein similarity scores from the training data, we further compute the semantic similarity scores with the gold programs using Dolos. Then the *aggregated similarity score* is computed as the maximum of the surface-level similarity score ($S_{\texttt{surface}}$) and semantic similarity score ($S_{\texttt{semantic}}$) similarity scores:

$$S(p, p^*) = \max(S_{\texttt{surface}}(p, p^*), S_{\texttt{semantic}}(p, p^*))$$

16. `https://tree-sitter.github.io/tree-sitter/`

17. This is determined by a combination of automatic and manual inspection. For example, at the 500^{th} most similar program from the Stack for MBPP, 95% of them have a similarity score < 72, which is no longer relevant by human inspection.

Benchmark	Models	Acc_o	Top-1 Score=100		Top-1 Score>90		Top-1 Score>80	
			% Rm	Acc_d	% Rm	Acc_d	% Rm	Acc_d
MBPP	StarCoderBase-15.5B	41.6	20.8	33.8 (-18.8%)	32.2	32.5 (-22.6%)	50.8	29.7 (-28.6%)
	Pythia-12B	17.8	3.6	17.0 (-4.5%)	6.9	16.6 (-6.7%)	11.4	15.8 (-11.2%)
	CodeGen-NL-16B	19.6	3.6	18.4 (-6.1%)	6.9	17.4 (-11.2%)	11.4	16.5 (-15.8%)
HumanEval	StarCoderBase-15.5B	30.5	18.9	22.6 (-25.9%)	39.6	15.2 (-50.2%)	63.4	20.0 (-34.4%)
	Pythia-12B	9.8	12.2	4.2 (-57.1%)	15.9	2.9 (-70.4%)	29.9	1.7 (-82.7%)
	CodeGen-NL-16B	14.6	12.2	8.3 (-43.2%)	15.9	5.8 (-60.3%)	29.9	3.5 (-76.0%)

Table 6.6: Measuring the de-contaminated accuracy (Acc_d) by removing potentially contaminated subsets of MBPP and HumanEval *w.r.t.* different thresholds. "Acc_o" denotes original model accuracy and "% Rm" denotes the percentage of the dataset removed. The *relative* accuracy degradation after de-contamination is shown in brackets.

This aggregated similarity score is a simple and intuitive way to reflect how programs can be similar both from their surface form and semantics.

6.4.2 Experimental Setup

We select two of the most popular public pretraining corpora for general LLM and code LLMs, namely the Pile (Gao et al., 2020) and the Stack (Kocetkov et al., 2022), and three series of popular open-source models, *i.e.,* Pythia (Biderman et al., 2023b), CodeGen-NL (Nijkamp et al., 2022) and StarCoderBase[18](Li et al., 2023). For the coding benchmark, we opt to study MBPP (Austin et al., 2021b) and HumanEval (Chen et al., 2021b) due to their popularity. We introduce them in more detail in the following subsection.

Models and Pretraining Data We select the models by the following criteria: *1)* The pretraining data for the models must be publicly available; *2)* To ensure non-trivial performance on the coding benchmarks, such models must have Python code in their pretraining data; *3)* Additionally, we do not consider any models that are instruction-tuned, or trained with reinforcement learning from human feedback (*i.e.,* RLHF), as it is hard to quantify the effect of such instruction-tuning/human preference data along with the pretraining corpus. Based on these criteria, we study the following three model series in this work:

The Pile *and Pythia.* Pythia (Biderman et al., 2023b) is a suite of 16 LLMs intended

18. StarcoderBase refers to the models originally trained on the Stack. These models were further finetuned on Python code to create the Starcoder models (Li et al., 2023).

to facilitate research in many areas. All models are trained on the PILE dataset (Gao et al., 2020), with their size ranging from 70M to 12B parameters. We used the 1.4B, 2.8B, 6.9B, and 12B models for this study. We use the GitHub split of the training dataset, which has a raw size of 95.16 GiB.

The PILE *and CodeGen-NL.* Another series of models that are trained with PILE is CodeGen-NL (Nijkamp et al., 2022), and we study the 350M, 2B, 6B, and 16B versions of it. Though stronger CodeGen models are available via further training on more code data, the exact copy of such data is not publicly released thus we choose to study the CodeGen-NL series. Due to the overlap of training data, we use the results of searching through the GitHub split that we did for the Pythia models.

The STACK *and StarCoderBase.* We use the 1B, 3B, 7B and 15.5B StarCoderBase models (Li et al., 2023) that were trained on the STACK dataset (Kocetkov et al., 2022). Due to the size of the training data, we only search through 60.40 GB within the Python split of its training dataset. The STACK was created from permissively-licensed source code files, and was open-sourced to make the training of code LLMs more reproducible.[19]

Benchmarks We measure the data contamination issues for the following two popular coding benchmarks:

MBPP (Austin et al., 2021b) is a benchmark containing 974 short, crowd-sourced Python programming problems. We use the 500 questions within its test split.

HumanEval (Chen et al., 2021b) is a benchmark consisting of 164 hand-written problems. Each problem contains a gold solution.

Notably, these two benchmarks come with gold program solutions, which we use to search the pretraining data as a query. To obtain the model performance and predictions on each of the dataset examples, we use the evaluation framework and model outputs from `L2CEval` (Ni et al., 2023b).

19. It is worth noticing that STACK went through a string-matching-based decontamination process for MBPP and HumanEval, but we are still able to find traces of contamination for these two datasets.

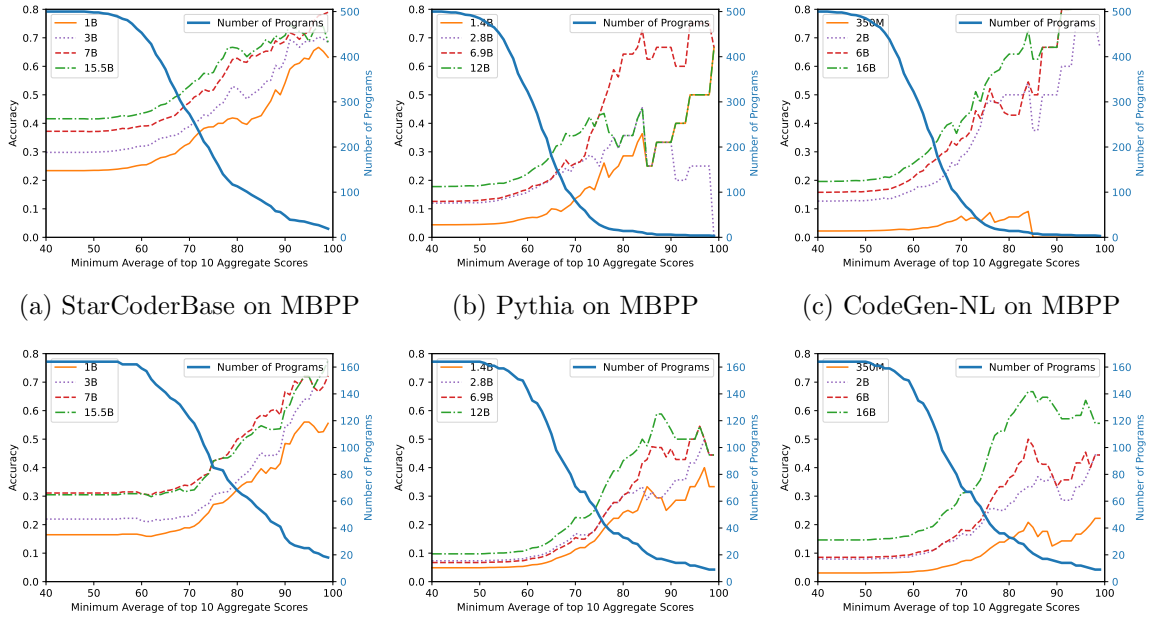
Models	MBPP			HumanEval		
	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$	$\Delta_{\Updownarrow}$	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$	$\Delta_{\Updownarrow}$
StarCoderBase	72.0	22.0	50.0	75.0	31.3	43.7
Pythia	40.0	8.0	42.0	56.3	0.0	56.3
CodeGen-NL	48.0	6.0	42.0	62.5	0.0	62.5

Table 6.7: We show the performance gap ($\Delta_{\Updownarrow}$) between the top 10% ($\Uparrow^{10\%}$) and bottom 10% ($\Downarrow_{10\%}$) of programs based on the average of the top-10 aggregated similarity scores. Only the *largest* models are shown for each model series.

6.4.3 Main Results

3.6% to 20.8% of the solutions are seen during training. For an example in the test data (*i.e.,* those of MBPP or HumanEval), we note it as "*seen*" if the aggregated similarity score is 100, *i.e.,* a perfect match exists on the surface- or semantic-level. Results show that 12.2% of the solutions in HumanEval have been seen by models trained on the Pile and 18.9% have been seen by models trained on the Stack. For MBPP, 3.6% of it can be found in the Pile while as much as 20.8% have been seen by models trained on the Stack. Much less overlap is found for the Pile, as 3.6% of MBPP, but 20.8% of the solutions on MBPP problems have been seen for models trained on the Stack. These results suggest that a non-trivial part of both MBPP and HumanEval have been seen for the models trained on either the Pile or the Stack, suggesting a high contamination rate.

Models perform significantly better when similar solutions are seen during training. To highlight the difference in performance that models have between questions which they have seen similar solutions and questions which they have not, following Razeghi et al. (2022), we use the performance gap between the top 10% most similar and bottom 10% least similar programs. The performance gap of models from the chosen model series is shown in Table. 6.7, where it can be observed that all three models perform significantly better on questions to which the solution is in the top 10% of compared to questions where the solution is in the bottom 10%. StarCoderBase-15.5B achieves an accuracy of 72% on the top 10% of questions and an accuracy of 22% on the bottom 10% of questions of the MBPP benchmark.



(a) StarCoderBase on MBPP (b) Pythia on MBPP (c) CodeGen-NL on MBPP

(d) StarCoderBase on HumanEval (e) Pythia on HumanEval (f) CodeGen-NL on HumanEval

Figure 6.7: Accuracy of different model series evaluated on a subset of examples with increasing overlap with the model's pretraining data. Subset obtained by using the x-axis as a threshold for the minimum score obtained by taking the average aggregated similarity score of top-10 matched programs in the training data. We note that as the number of examples decreases, it becomes more likely for the lines to overlap, as can be seen in Figure. 6.7b.

De-contaminated results. In an attempt to show the impact of seen questions on model performance, we remove potentially contaminated questions from each benchmark, showing the results in Table. 6.6. We observe that removing not only questions that have been seen, but also questions where programs similar to the gold program have been seen during training has an adverse effect on model performance. Moreover, from the de-contaminated results, the performance gap between different models could be much smaller. For example, the original accuracy (Acc_o) gap between StarCoderBase-15.5B and Pythia-12B is 23.8%, and after de-contamination, the performance gap is decreased to 13.9%. This indicates that a large part of the performance gap between different models may due to data contamination. While we do not find the performance rankings of the models to change with de-contaminated results, a study with more models might be needed for deriving any general conclusions.

6.5 Detailed Experiment Settings

6.5.1 Example Model Inputs and Outputs

To better understand the inputs and outputs of the models for different tasks, here we show how we unify different language-to-code generation tasks in Table. 6.8.

Instruction	Programming Context	Natural Language	Code
Text-to-SQL Parsing (Spider, WikiTQ)			
Given database schema and a question in natural language, generate the corresponding SQL query.	-- Database game_injury: -- Table stadium: id, name, Home_Games, Average_Attendance, Total_Attendance, Capacity_Percentage -- Table game: stadium_id, id, Season, Date, Home_team, Away_team, Score, Competition -- Table injury_accident: game_id, id, Player, Injury, Number_of_matches, Source	How many distinct kinds of injuries happened after season 2010?	SELECT count(DISTINCT T1. Injury) FROM injury_accident AS T1 JOIN game AS T2 ON T1. game_id = T2.id WHERE T2.Season > 2010
Math Reasoning (GSM8k, SVAMP)			
Given questions in the comment, use python programs to produce the correct answers with the ` answer' variable.	-	James takes 2 Tylenol tablets that are 375 mg each, every 6 hours. How many mg does he take a day?	mg_tylenol_per_tablet = 375 mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet hours_per_day = 24 times_per_day = hours_per_day / 6 mg_each_day = mg_tylenol_taken_each_time * times_per_day answer = mg_each_day
Python Programming (MBPP)			

Given the natural language description and example assertion(s), write a python function.	# These are the assertions for your function:
	assert similar_elements ((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)
		Write a function to find the similar elements from the given two tuple lists.	def similar_elements(test_tup1, test_tup2):
			  res = tuple(set(test_tup1) & set(test_tup2))
			  return (res)

Table 6.8: Example programming context, natural language input, and output code for all tasks. The instructions are fixed for different examples in the same task. Full prompts are shown in § 6.10.

6.5.2 Task-specific Setups

Here we discuss the implementation details for each task.

Spider. For Spider, we follow previous work (Ni et al., 2023a; Rajkumar et al., 2022) and add database schema as part of the prompt so that the LLMs are able to ground the language input onto the specific tables and columns. We use the official evaluation script[20] to obtain the execution accuracy, by comparing the execution results of the predicted and gold SQL query;

WikiTQ. In addition to adding database schema, for WikiTQ, we also follow (Lin et al., 2020) and add a non-empty example value next to each column. This is because WikiTQ mostly consists of noisy web tables, thus adding example values will help the model better understand the semantics of the columns;

GSM8k and SVAMP. For GSM8k and SVAMP, we follow (Chen et al., 2022b; Ni et al., 2022) and generate idiomatic programs (*i.e.,* programs with meaningful variable names) as solutions. Those programs are later executed and the variable "`answer`" will be used as the final answer[21];

MBPP. For MBPP, three assertions are given for each example to verify the correctness of the generated programs. Same as (Shi et al., 2022a; Zhang et al., 2022), we use one of the assertions as the input (*i.e.,* open test case) to prompt the model so it would have the information of the function signature, and only when the generated program passes all three assertions (including the rest two, which can be seen as hidden tests) do we count execution accuracy to be 1;

HumanEval. To allow direct comparison with previous work (Chen et al., 2021a; Li et al., 2023; Nijkamp et al., 2022), we use the function header and docstrings as the context to prompt the model to generate a completion of the function;

20. `https://github.com/taoyds/spider`

21. In zero-shot experiments, the instruction also clearly states this as in Table. 6.8

DS-1000. We follow the original paper (Lai et al., 2022) to prompt the model and generate Python lines that complete the functionality described in natural language.

6.5.3 Details for Selected Models

Here we provide more detailed descriptions of the models that we evaluate in this work.

OpenAI models. We directly use the engine name from OpenAI API documentation in this work to avoid any confusion in naming. Though much information about those models is opaque, we do know that codex-cushman-001 corresponds to the 12B model described in (Chen et al., 2021a) and that text-davinci-002 is an InstructGPT model based on code-davinci-002 [22];

CodeGen models. CodeGen (Nijkamp et al., 2022) is a series of models that are trained through multiple stages, ranging from 2B to 16B sizes. The models are first trained on the Pile dataset (Gao et al., 2020) which contains mostly text data with some mixture of code, yielding the CodeGen-nl version. Then it is further pretrained on the BigQuery (Bisong and Bisong, 2019) and BigPython (Nijkamp et al., 2022) data, obtaining the CodeGen-multi and CodeGen-mono versions, respectively;

EleutherAI models and Dolly. Based on the architecture of GPT-NeoX (Black et al., 2022), EleutherAI devotes to creating open-source replications of GPT-3 by training on the Pile (Gao et al., 2020). The latest model series, Pythia (Biderman et al., 2023a), includes a set of models ranging from 1.4B to 12B, with 154 intermediate checkpoints also released. Dolly (Conover et al., 2023) is a series of models instruction-tuned from Pythia with the databricks-dolly-15k[23] instructional data from Databricks;

BigCode and Replit models. BigCode is a project aiming to create open-source language models with strong code generation abilities. Trained on different versions of the Stack (Kocetkov et al., 2022), SantaCoder (Allal et al., 2023) and StarCoder (Li et al., 2023) are

22. `https://platform.openai.com/docs/model-index-for-researchers`

23. `https://github.com/databrickslabs/dolly`

two models with different sizes (*i.e.,* 1.1B and 15.5B), with StarCoder being comparable to the OpenAI's code-cushman-001 model;

LLaMA and Alpaca. LLaMA (Touvron et al., 2023a) is a series of models pretrained to be compute-optimal during inference, with performance close to GPT-3.5 models on various academic NLP tasks. And Alpaca (Taori et al., 2023) is its instruction-tuned version with 52K instruction-following data distilled from OpenAI's text-davinci-003. We use the Alpaca-LoRA version[24] in this work.

6.6 Additional Results

6.6.1 Full Few-shot Results

Here we show the full few-shot results for all 56 models across different tasks for reference. Following (Liang et al., 2022), per-dataset win rates are first computed by head-to-head model comparison on each dataset, then the mean win rates are calculated by taking the average of the per-dataset win rate. For the number of shots (*i.e.,* exemplars) in the prompt, we use 2 for Spider and WikiTQ, 8 for GSM8k, and 3 for MBPP. This distinction is to be maximally comparable with previous work for each dataset, as well as accommodating models with smaller context lengths.

6.6.2 Full Quantitative Analysis

Additional error analysis for Spider. In § 6.3.6, we only discussed the common error modes for math reasoning and Python programming across different models. Here we also show the error analysis for text-to-SQL parsing, using Spider as the representative dataset in Table. 6.10a. From the results, we can see that for Spider, the main differentiating factor for different models lies in the execution errors. Upon inspection, this is not because the models are generating deformed SQL queries with grammatical errors, but because the models failed to understand the database schema.

24. `https://github.com/tloen/alpaca-lora`

Organization	Model Name	Spider (2)	WikiTQ (2)	GSM8k (8)	SVAMP (4)	MBPP (3)	HE (0)	DS1K (0)
OpenAI	code-cushman-001	49.6	23.8	21.8	–	42.6	–	–
	code-davinci-002	73.7	47.2	68.1	–	61.1	–	–
	text-davinci-002	67.7	44.8	59.9	77.4	56.8	16.5	16.2
	text-davinci-003	68.3	45.4	64.1	80.7	63.6	52.4	15.3
	gpt-3.5-turbo-0301	72.7	38.4	74.7	80.9	66.6	24.4	15.9
	gpt-3.5-turbo-0613	73.6	44.6	66.7	80.7	67.4	39.0	11.0
	gpt-4-0314	77.2	56.2	92.4	92.4	74.0	76.8	24.9
	gpt-4-0613	79.2	56.7	88.5	92.8	74.2	80.5	24.0
Meta AI	InCoder-1B	13.4	6.2	1.0	3.5	13.8	8.5	2.9
	InCoder-6B	24.1	13.3	3.1	9.4	20.4	15.9	5.4
	LLaMA-7B	13.1	10.3	8.0	34.4	16.6	11.0	3.7
	LLaMA-2-7B	21.7	14.3	12.7	36.4	21.2	11.0	5.3
	CodeLLaMA-7B	54.3	29.5	25.5	52.8	40.0	31.1	16.0
	LLaMA-13B	15.2	15.7	15.7	44.7	22.8	12.2	6.5
	LLaMA-2-13B	35.7	24.6	26.1	58.9	27.0	17.7	9.1
	CodeLLaMA-13B	58.5	35.6	30.7	64.9	44.0	34.2	18.8
	LLaMA-30B	38.5	30.5	15.9	57.6	26.6	21.3	7.6
	CodeLLaMA-34B	61.7	32.3	43.6	70.7	45.6	44.5	22.4
	LLaMA-65B	43.2	26.8	18.9	65.5	32.1	23.2	7.5
	LLaMA-2-70B	58.5	37.3	54.3	73.9	26.6	28.7	16.9
Salesforce	CodeGen-6B-multi	15.7	8.4	5.0	23.0	21.0	21.3	2.1
	CodeGen-6B-mono	33.1	16.4	8.1	23.8	35.0	27.4	7.1
	CodeGen-16B-multi	24.6	13.1	7.1	27.7	24.4	20.1	6.2
	CodeGen-16B-mono	35.1	16.5	12.8	31.2	37.8	32.3	9.2
	CodeGen2.5-7B-multi	53.8	29.6	14.9	43.1	38.2	31.1	16.9
	CodeGen2.5-7B-mono	39.2	26.5	13.5	38.7	46.0	31.7	11.4
	CodeGen2.5-7B-instruct	44.1	23.4	17.8	42.1	45.8	37.2	14.1
	XGen-7B-8k-base	28.3	17.9	7.1	32.8	20.8	13.4	6.3
EleutherAI	Pythia-1.4B	5.7	4.4	1.5	9.3	5.8	3.7	1.8
	Pythia-6.9B	12.5	7.2	2.6	21.4	13.2	9.8	2.3
	Pythia-12B	16.2	14.3	2.6	20.8	19.0	11.0	2.0
	GPT-J-6B	20.3	12.7	2.3	14.5	14.6	9.1	3.1
	GPT-NeoX-20B	24.6	17.0	5.8	28.8	19.0	14.0	4.2
DataBricks	dolly-v2-7B	13.1	10.6	2.6	12.7	12.0	7.3	2.9
	dolly-v2-12B	13.0	6.8	2.6	12.7	3.8	9.7	1.8
BigCode	SantaCoder-1B	19.0	11.4	2.8	0.0	26.2	17.7	1.1
	StarCoder-15.5B	52.1	27.4	22.1	48.8	46.6	34.2	19.8
	StarChat-15.5B	54.1	28.7	19.5	48.7	42.8	29.3	19.6
	StarCoderPlus	47.9	27.7	25.1	54.1	36.6	25.6	16.1
Stanford	Alpaca-LoRA-7B	16.1	12.0	3.5	23.7	14.4	8.5	4.5
	Alpaca-LoRA-13B	24.3	25.4	18.5	47.8	23.4	15.9	8.0
	Alpaca-LoRA-30B	46.2	39.7	19.4	64.8	32.0	23.8	11.3
WizardLM	WizardCoder-15B	58.6	29.4	25.8	56.1	47.4	51.2	20.8
Replit	replit-code-v1-3b	39.3	28.3	5.6	24.2	30.6	21.3	7.3
LMSYS	Vicuna-7B-v1.5	9.3	11.2	10.8	39.3	17.6	17.7	5.9
	Vicuna-13B-v1.3	29.3	13.6	18.1	46.9	20.8	18.9	8.5
	Vicuna-33B-v1.3	37.9	24.7	4.8	57.0	27.8	20.1	5.6
MosaicML	MPT-7B	27.3	16.4	10.9	34.8	21.0	14.0	7.6
	MPT-7B-instruct	25.5	18.4	10.4	36.9	24.0	12.8	7.2
	MPT-30B	43.3	24.6	30.7	57.9	29.2	22.0	12.5
	MPT-30B-instruct	42.8	20.0	29.1	57.9	28.4	22.6	9.3
XLang	Lemur-70b	68.0	44.9	57.5	47.9	51.4	–	–
Mistral AI	Mistral-7B-v0.1	53.3	31.4	38.4	69.4	37.8	25.0	14.1
	Mistral-7B-v0.1-instruct	39.0	40.5	34.0	27.4	58.3	27.4	8.8

Table 6.9: Full few/zero-shot learning results for all models. Number in "(·)" denotes the number of shots (*i.e.,* exemplars) in the prompt. -: result not available yet.

Analysis when model produces correct programs. In Table. 6.10b, we also give an analysis of the correct programs that the model generates. More specifically, we categorize them into three cases: 1) when they are spurious, *i.e.,* achieve the correct execution result by chance; 2) when they are the same as the reference program[25]; and 3) when the generated program explores a different path than the gold program. From the results, we can see that the spuriousness problem varies for different tasks, as there are non-trivial percentages (*i.e.,* ∼7%) of spurious programs for Spider, but almost none observed for GSM8k and MBPP. We think this is because for Spider, the execution results are more generic numbers or cell values, which are easier for an incorrect program to execute to by chance. Moreover, we can also see that across all three tasks, the models are often able to generate correct programs that are different from the gold ones. This suggests that the models may benefit from self-bootstrapping methods such as (Ni et al., 2022).

6.6.3 Full Calibration Evaluation Results

Following previous work (Liang et al., 2022), we measure model calibration based on two metrics, ECE (expected calibration error) and SCAA (selective coverage-accuracy area. In § 6.3.5 we showed the calibration results with ECE and here we show both calibration metrics with the execution accuracy and the model rankings with respect to all these three metrics. From the results, we can see that while ECE shows that a highly accurate model can also be poorly calibrated, SCAA is much more correlated with execution accuracy. This is because the calculation of ECE is independent of the model performance (*i.e.,* accuracy), and SCAA, which is based on selective classification, is positively impacted by the model performance.

6.6.4 Scaling Curves for Each Task

Here in Figure. 6.8, we show the performance of the models with respect to their sizes on each task. While the consistent trend is that larger models are generally better for all tasks, the scaling law slightly varies for different tasks. For tasks that are more demanding for

25. Here we evaluate semantic equivalence by manual inspection and not exact string match.

Dataset	Models	Exec. Err.	Missing S.	Extra S.	Wrong S.	Unclear NL	Total
Spider	StarCoder-15.5B	17	2	5	15	3	42
	code-cushman-001	32	2	4	13	3	54
	code-davinci-002	7	4	4	8	3	26
	gpt-4	2	4	7	11	2	26
GSM8k	StarCoder-15.5B	12	11	4	52	1	80
	code-cushman-001	11	11	4	50	3	79
	code-davinci-002	4	15	4	13	5	41
	gpt-4	1	1	0	3	1	6
MBPP	StarCoder-15.5B	6	10	1	23	12	52
	code-cushman-001	9	19	2	25	11	57
	code-davinci-002	5	10	2	13	9	34
	gpt-4	3	8	1	8	8	25

(a) Error analysis when models fail to produce the correct programs.

Dataset	Models	Spurious	Same as Gold	Different from Gold	Total
Spider	StarCoder-15.5B	7	36	15	58
	code-cushman-001	6	23	17	46
	code-davinci-002	7	43	24	74
	gpt-4	7	35	32	74
GSM8k	StarCoder-15.5B	1	15	4	20
	code-cushman-001	0	18	3	21
	code-davinci-002	1	45	13	59
	gpt-4	0	78	16	94
MBPP	StarCoder-15.5B	1	17	30	48
	code-cushman-001	1	13	29	43
	code-davinci-002	0	22	44	66
	gpt-4	0	21	54	75

(b) Analysis when models produce programs that are evaluated to be correct.

Table 6.10: Full quantitative analysis results via manual inspection of the model outputs. "Missing/Extra/Wrong S." denote Missing/Extra/Wrong Steps, respectively.

programming skills (*e.g.,* Spider, MBPP), the scaling is relatively smooth and for tasks that require more language understanding and reasoning (*e.g.,* WikiTQ, GSM8k), the trend appears to be more emergent (Wei et al., 2022a).

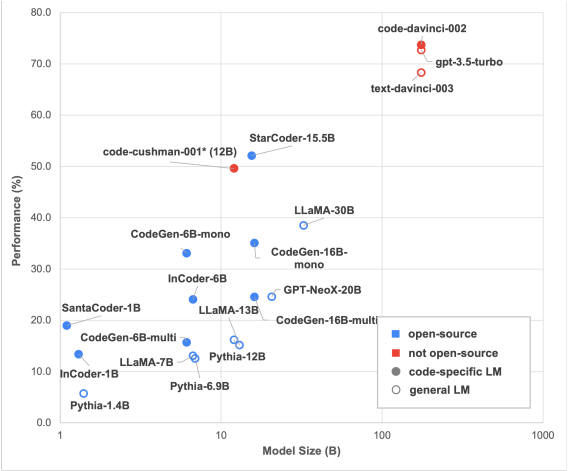
6.7 Limitations

While we strive to provide a comprehensive and fair evaluation of LLMs in `L2C` tasks, here we also discuss the limitations of `L2CEval`.

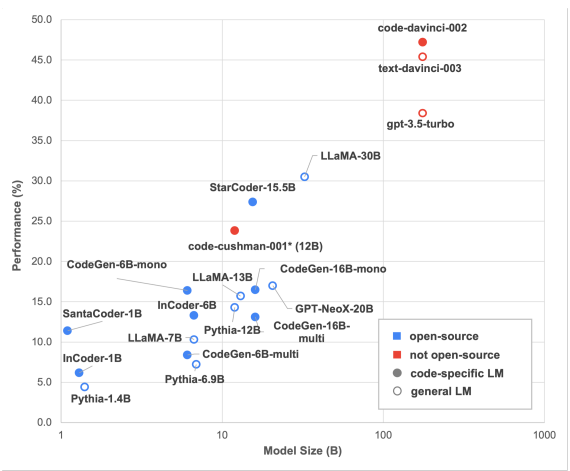
Generation using greedy-decoding. In this work, we use greedy decoding to generate a single program for each example as the models' output. While this is the most efficient

Model Names	Exec. Acc. (↑)		ECE (↓)		SCAA (↑)	
	Value	Rank	Value	Rank	Value	Rank
StarCoder-15.5B	37.1	1	0.629	3	0.371	2
code-cushman-001	34.5	2	0.496	1	0.431	1
Alpaca-30B	34.3	3	0.684	5	0.324	3
replit-code-v1-3b	26.0	4	0.594	2	0.301	4
CodeGen-16B-mono	25.6	5	0.739	6	0.284	5
CodeGen-6B-mono	23.2	6	0.762	7	0.261	6
Alpaca-13B	22.9	7	0.787	8	0.221	7
SantaCoder-1B	14.9	8	0.643	4	0.201	8
Pythia-12B	13.0	9	0.879	10	0.135	9
LLaMA-7B	12.0	10	0.879	9	0.132	10
Alpaca-7B	11.5	11	0.884	11	0.124	11

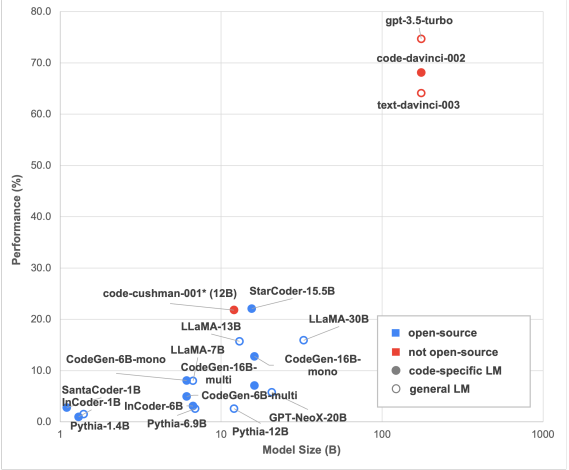
Table 6.11: Full calibration results. All metrics are average across all tasks. ECE denotes *expected calibration error* and SCAA denotes *selective coverage-accuracy area*. "↑ / ↓" means higher/lower is better.



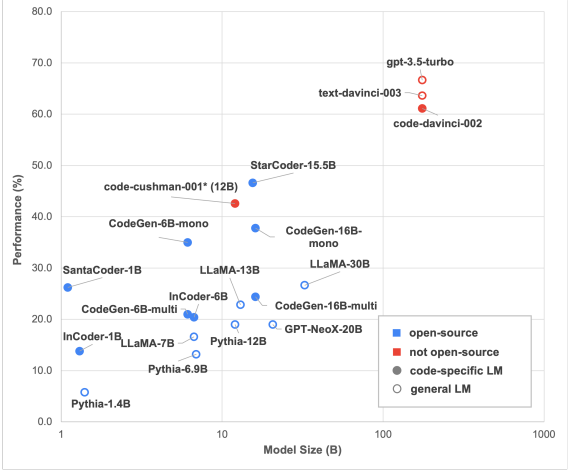
(a) Spider



(b) WikiTQ



(c) GSM8k



(d) MBPP

Figure 6.8: Model size scaling for each task.

way of generation and ensures fair comparison for different models as it is not affected by factors like sampling temperature, it is also relatively noisy (Chen et al., 2021a; Nijkamp et al., 2022). For tasks such as MBPP or Python programming in general, sampling k solutions then measure *pass@k* (any of the k programs being correct) or *n@k* (*i.e.,* # the k programs being correct) are better as they give the model k tries to generate the correct program to lower the variance. For Python programming tasks, such methods are closer to practical use cases as we typically have test cases that can filter out some incorrect programs in the samples. For other tasks, having a better *pass@k* also provides opportunities for post-generation reranking methods such as (Ni et al., 2023a; Shi et al., 2022a; Zhang et al., 2022). However, the cost for evaluating *pass@k* or *n@k* is k times of the compute compared with greedy decoding, thus we choose to only evaluate greedy decoding in this work and leave sampling-based evaluation to future work.

Execution-based evaluation. Moreover, we mainly rely on execution-based evaluation (*i.e.,* execution accuracy) for this work. However, such evaluation may produce spurious programs, *i.e.,* false-positive programs that achieve the correct execution result by chance (Ni et al., 2020; Zhong et al., 2020). In this work, we adopt human evaluation to measure the problem of spuriousness and found non-trivial portion of "correct" programs being spurious for Spider but not for other datasets. More details on this can be found in § 6.6.2. In addition, execution may not always be straightforward in practice, especially when complex dependencies and potentially harmful programs are considered (Chen et al., 2021a).

Confounding factors during comparison. When comparing models, especially across different model series, there are typically multiple performance-impacting factors that are in effect at the same time, some of which are not studied in this work, such as model architecture and pretraining objective. Such confounding factors may limit the validity of the conclusions that we draw from model comparisons. In this work, we try to mitigate this by fixing as many variables about the models as possible during a comparison, such as making observations within the same model series. While the general trend can still be observed across different model series, we should also note that when interpreting the results,

readers should be mindful of such confounding factors when comparing different models.

Lack of information for proprietary models. For the open-access proprietary LLMs (*e.g.,* OpenAI models), due to the lack of basic information and mismatches between the models described in the papers and the actual API engines, very few scientific conclusions can be drawn from these results. We evaluate such proprietary models mainly to provide baselines and in the hope of helping practitioners in choosing models for their use cases. We also present human evaluations of some of the strong models to discuss differences in common error modes. However, when making our findings, we generally rely on open-source models instead, to avoid being misled by speculative model details of such closed-source models.

6.8 Related Work

Code generation evaluation. Several code generation benchmarks are collected from raw data from GitHub and StackOverflow, and involve professional annotators to enhance the quality of the data (Agashe et al., 2019a; Iyer et al., 2018; Yin et al., 2018a). While such benchmarks focus more on lexical-based evaluation, ODEX (Wang et al., 2022d) introduces execution-based evaluation, which has also been widely applied in recent code generation evaluation benchmarks, such as DS-1000 (Lai et al., 2022), HumanEval (Chen et al., 2021a), and MBPP (Austin et al., 2021a). More recently, there has been an increasing focus on assessing the generalization capabilities of code generation models across multiple programming languages (Athiwaratkun et al., 2023), and benchmarks such as CodeGeeX (Zheng et al., 2023) and MultiPL-E (Cassano et al., 2023). In our work, we focus on studying whether LLMs can map natural language instructions to code using the *most popular* programming languages for each domain (*i.e.,* SQL for semantic parsing and Python for math reasoning and programming). While the study of different programming languages are orthogonal to our work, we refer the readers to these existings works on multi-lingual evaluation benchmarks.

Other code-related tasks. Large language models have also shown significant success in other code-related directions. One popular direction is code understanding. For example, CodeXGLUE (Lu et al., 2021b) comprises three widely-used code understanding tasks including defect detection, clone detection, and code search. However, CONCODE (Iyer et al., 2018) is the only language-to-code task included in CodeXGLUE and it uses surface-form based evaluation metrics such as BLEU. BigCloneBench (Krinke and Ragkhitwetsagul, 2022) tasks to measure the similarity between code pairs to predict whether they have the same functionality. CodeSearchNet (Husain et al., 2019) is a benchmark of semantic code search given natural language queries. Besides code understanding, there have been other tasks such as code translation (Lachaux et al., 2020) and program repair (Gupta et al., 2017). We leave systematic evaluation of LLMs on those tasks as important future work.

6.9 Summary

In this chapter, we present `L2CEval`, a comprehensive evaluation framework for natural language to code generation, and we evaluate 56 models from 13 organizations, on 7 tasks from 3 core domains. `L2CEval` investigates models' performance on a variety of axes such as model scale, training data mixture, sensitivity to few-shot exemplars as well as the impact of instruction tuning, *inter alia*. We also present an analysis on the model calibration and conduct a human evaluation of common error modes across different models. We hope our study will provide useful insights for the community into applying LLMs for downstream code applications and future model development efforts.

6.10 Appendix – Full Prompts

Here we append the prompts that we use for few-shot experiments for Spider (Table. 6.12), WikiTQ (Table. 6.13), GSM8k (Table. 6.14 and Table. 6.15), SVAMP (Table. 6.16, MBPP (Table. 6.17), HumanEval (Table. 6.18), and DS-1000 (Table. 6.19).

```
-- Given database schema and a question in natural language, generate the corresponding SQL query.

-- Example:

-- Database game_injury:
--  Table stadium: id, name, Home_Games, Average_Attendance, Total_Attendance, Capacity_Percentage
--  Table game: stadium_id, id, Season, Date, Home_team, Away_team, Score, Competition
--  Table injury_accident: game_id, id, Player, Injury, Number_of_matches, Source
-- Question: How many distinct kinds of injuries happened after season 2010?
-- SQL:
SELECT count(DISTINCT T1.Injury) FROM injury_accident AS T1 JOIN game AS T2 ON T1.game_id  =  T2.id
WHERE T2.Season  >  2010

-- Example:

-- Database farm:
--  Table city: City_ID, Official_Name, Status, Area_km_2, Population, Census_Ranking
--  Table farm: Farm_ID, Year, Total_Horses, Working_Horses, Total_Cattle, Oxen, Bulls, Cows, Pigs,
Sheep_and_Goats
--  Table farm_competition: Competition_ID, Year, Theme, Host_city_ID, Hosts
--  Table competition_record: Competition_ID, Farm_ID, Rank
-- Question: Return the hosts of competitions for which the theme is not Aliens?
-- SQL:
SELECT Hosts FROM farm_competition WHERE Theme !=  'Aliens'

-- Example:

-- Database concert_singer:
--  Table stadium: Stadium_ID, Location, Name, Capacity, Highest, Lowest, Average
--  Table singer: Singer_ID, Name, Country, Song_Name, Song_release_year, Age, Is_male
--  Table concert: concert_ID, concert_Name, Theme, Stadium_ID, Year
--  Table singer_in_concert: concert_ID, Singer_ID
-- Question: Show name, country, age for all singers ordered by age from the oldest to the youngest.
-- SQL:

SELECT Name, Country, Age FROM singer ORDER BY Age DESC
```

Table 6.12: The `prompt` we use for the Spider dataset, with an example input and gold output.

```
-- Given database schema and a question in natural language, generate the corresponding SQL query.

-- Example:

-- Database 204_126:
--  Table main_table: id (1), agg (0), place (t1), place_number (1.0), player (larry nelson), country
(united states), score (70-72-73-72=287), score_result (287), score_number (70), score_number1 (70),
score_number2 (72), score_number3 (73), score_number4 (72), to_par (-1), to_par_number (-1.0),
money_lrb_rrb (playoff), money_lrb_rrb_number (58750.0)
-- Question: what was first place 's difference to par ?
-- SQL:
select to_par from main_table where place_number = 1

-- Example:

-- Database 204_522:
--  Table main_table: id (1), agg (0), boat_count (4911), boat_count_number (4911), boat_count_minimum
 (4951), boat_count_maximum (4955), name (ha-201), builder (sasebo naval arsenal), laid_down
(01-03-1945), laid_down_number (1), laid_down_parsed (1945-01-03), laid_down_year (1945),
laid_down_month (1), laid_down_day (3), launched (23-04-1945), launched_number (23), launched_parsed
(1945-04-23), launched_year (1945), launched_month (4), launched_day (23), completed (31-05-1945),
completed_number (31), completed_parsed (1945-05-31), completed_year (1945), completed_month (5),
completed_day (31), fate (decommissioned 30-11-1945. scuttled off goto islands 01-04-1946)
-- Question: when was a boat launched immediately before ha-206 ?
-- SQL:
select name from main_table where launched_parsed < ( select launched_parsed from main_table where
name = 'ha-206' ) order by launched_parsed desc limit 1

-- Example:

-- Database 204_706:
--  Table main_table: id (1), agg (0), year (1996), year_number (1996), competition (world junior
 championships), venue (sydney, australia), position (15th (q)), position_first (15th),
 position_second (q), position_first_number (15.0), notes (7.43 m), notes_first (7.43 m),
 notes_second (w), notes_first_number (7.43), notes_second_result (w)
--  Table t_venue_address: m_id (1), venue_address (sydney)
-- Question: what was the venue when he placed first ?
-- SQL:

select venue from main_table where position_first = '1st'
```

Table 6.13: The `prompt` we use for the WikiTQ dataset, with an example input and gold output.

```
## Given questions in the comment, use python programs to produce the correct answers with the `answer
` variable.

## Cristina, John, Clarissa and Sarah want to give their mother a photo album for her birthday.
Cristina brings 7 photos, John brings 10 photos and Sarah brings 9 photos. If the photo album has 40
slots available, how many photos does Clarissa need to bring in order to complete the photo album?
n_photo_cristina = 7
n_photo_john = 10
n_photo_sarah = 9
n_photo_total = n_photo_cristina + n_photo_john + n_photo_sarah
n_slots = 40
n_slots_left = n_slots - n_photo_total
answer = n_slots_left

## Katy, Wendi, and Carrie went to a bread-making party.  Katy brought three 5-pound bags of flour.
Wendi brought twice as much flour as Katy, but Carrie brought 5 pounds less than the amount of flour
Wendi brought.  How much more flour, in ounces, did Carrie bring than Katy?
pound_flour_katy = 3 * 5
pound_flour_wendi = pound_flour_katy * 2
pound_flour_carrie = pound_flour_wendi - 5
pound_diff_carrie_katy = pound_flour_carrie - pound_flour_katy
ounce_diff_carrie_katy = pound_diff_carrie_katy * 16
answer = ounce_diff_carrie_katy

## James takes 2 Tylenol tablets that are 375 mg each, every 6 hours.  How many mg does he take a day?
mg_tylenol_per_tablet = 375
mg_tylenol_taken_each_time = 2 * mg_tylenol_per_tablet
hours_per_day = 24
times_per_day = hours_per_day / 6
mg_each_day = mg_tylenol_taken_each_time * times_per_day
answer = mg_each_day

## Kyle bakes 60 cookies and 32 brownies. Kyle eats 2 cookies and 2 brownies. Kyle's mom eats 1 cookie
 and 2 brownies. If Kyle sells a cookie for $1 and a brownie for $1.50, how much money will Kyle make
if he sells all of his baked goods?
n_cookies = 60
n_brownies = 32
n_cookies_left_after_kyle = n_cookies - 2
n_brownies_left_after_kyle = n_brownies - 2
n_cookies_left_after_kyle_mom = n_cookies_left_after_kyle - 1
n_brownies_left_after_kyle_mom = n_brownies_left_after_kyle - 2
money_earned_kyle = n_cookies_left_after_kyle_mom * 1 + n_brownies_left_after_kyle_mom * 1.5
answer = money_earned_kyle

## There were 63 Easter eggs in the yard.  Hannah found twice as many as Helen.  How many Easter eggs
did Hannah find?
n_easter_eggs = 63
unit_times = 2
total_units = unit_times + 1
n_easter_eggs_per_unit = n_easter_eggs / total_units
n_easter_eggs_helen = n_easter_eggs_per_unit * 1
n_easter_eggs_hannah = n_easter_eggs_per_unit * 2
answer = n_easter_eggs_hannah

## Ethan is reading a sci-fi book that has 360 pages. He read 40 pages on Saturday morning and another
 10 pages at night. The next day he read twice the total pages as on Saturday. How many pages does he
have left to read?
n_pages = 360
total_page_saturday = 40 + 10
total_page_next_day = total_page_saturday * 2
total_pages_read = total_page_saturday + total_page_next_day
n_pages_left = n_pages - total_pages_read
answer = n_pages_left
```

Table 6.14: The `prompt` we use for the GSM8k dataset, with an example input and gold output. (Part 1)

```
## A library has a number of books. 35percent_books_for_children = 0.35
percent_books_for_adults = 1.0 - percent_books_for_children
n_books_for_adults = 104
n_books_in_total = n_books_for_adults / percent_books_for_adults
answer = n_books_in_total

## Tyler has 21 CDs. He gives away a third of his CDs to his friend. Then he goes to the music store
and buys 8 brand new CDs. How many CDs does Tyler have now?
n_cds_tyler = 21
percent_cds_given_away =  1.0 / 3.0
n_cds_left_after_giving_away = n_cds_tyler - n_cds_tyler * percent_cds_given_away
n_new_cds_purchased = 8
n_cds_now = n_cds_left_after_giving_away + n_new_cds_purchased
answer = n_cds_now

## Matt has six cats and half of them are female. If each female cat has 7 kittens, and Matt sells 9
 of them, what percentage of his remaining cats are kittens (rounded to the nearest percent)?

n_cats_matt = 6
n_female_cats = n_cats_matt / 2
n_kittens_per_female_cat = 7
n_kittens_total = n_female_cats * n_kittens_per_female_cat
n_kittens_sold = 9
n_kittens_left = n_kittens_total - n_kittens_sold
n_cats_total = n_cats_matt + n_kittens_left
percent_kittens = (n_kittens_left / n_cats_total) * 100
answer = round(percent_kittens)
```

Table 6.15: The `prompt` we use for the GSM8k dataset, with an example input and expected output. (Part 2)

```
## Given questions in the comment, use python programs to produce the correct answers with the `answer
` variable.

## A waiter had some customers. After 9 customers left he still had 12 customers. How many customers
did he have at the start?
n_customers_left = 9
n_customers_now = 12
n_customers_start = n_customers_now + n_customers_left
answer = n_customers_start

## 3 birds were sitting on the fence. 6 more storks and 2 more birds came to join them. How many more
storks than birds are sitting on the fence?
n_birds = 3
n_storks = 6
n_more_bird = 2
n_more_stork_than_bird = n_storks - (n_birds + n_more_bird)
answer = n_more_stork_than_bird

## They decided to hold the party in their backyard. If they have 11 sets of tables and each set has
13 chairs. How many chairs do they have in the backyard?
n_tables = 11
n_chairs_per_table = 13
n_chairs = n_tables * n_chairs_per_table
answer = n_chairs

## 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?
group_size = 18
n_total_bananas = 180
n_groups = n_total_bananas / group_size
answer = n_groups

## In a school there are 697 girls and the rest are boys. If there are 228 more girls than boys. How
 many boys are there in that school?

n_girls = 697
n_more_girls = 228
n_boys = n_girls - n_more_girls
answer = n_boys
```

Table 6.16: The `prompt` we use for the SVAMP dataset, with an example input and gold output.

```
## Given the natural language description and example assertion(s), write a python function.

### Task Start ###
# These are the assertions for your function:
assert similar_elements((3, 4, 5, 6),(5, 7, 4, 10)) == (4, 5)

""" Write a function to find the similar elements from the given two tuple lists. """
def similar_elements(test_tup1, test_tup2):
    res = tuple(set(test_tup1) & set(test_tup2))
    return (res)
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert is_not_prime(2) == False

""" Write a python function to identify non-prime numbers. """
import math
def is_not_prime(n):
    result = False
    for i in range(2,int(math.sqrt(n)) + 1):
        if n result = True
    return result
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert heap_queue_largest( [25, 35, 22, 85, 14, 65, 75, 22, 58],3)==[85, 75, 65]

""" Write a function to find the largest integers from a given list of numbers using heap queue
algorithm. """
import heapq as hq
def heap_queue_largest(nums,n):
    largest_nums = hq.nlargest(n, nums)
    return largest_nums
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert find_Volume(10,8,6) == 240

""" Write a python function to find the volume of a triangular prism. """

def find_Volume(l,b,h) :
    return ((l * b * h) / 2)
### Task End ###
```

Table 6.17: The `prompt` we use for the MBPP dataset, with an example input and expected output.

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """

    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

Table 6.18: An example `prompt` we use for the HumanEval dataset with input and expected output.

```
Problem:

Can you give me any suggestion that transforms a sklearn Bunch object (from sklearn.datasets) to a
 dataframe? I'd like to do it to iris dataset.
Thanks!

from sklearn.datasets import load_iris
import pandas as pd
data = load_iris()
print(type(data))
data1 = pd. # May be you can give me a Pandas method?

A:

<code>
import numpy as np
from sklearn.datasets import load_iris
import pandas as pd
data = load_data()
</code>
data1 = ... # put solution in this variable
BEGIN SOLUTION
<code>

data1 = pd.DataFrame(data=np.c_[data['data'], data['target']], columns=data['feature_names'] + ['
 target'])
```

Table 6.19: An example `prompt` we use for the DS-1000 dataset with input and expected output.

Chapter 7

Conclusion and Future Work

In this dissertation, we address several challenges for applying language models for the task of program synthesis from natural language.

In Chapter 3, we show that it is beneficial for the language models to learn from their own program samples that are also correct, or even partially-correct for each natural language input. While experiments are mostly conducted on math reasoning domain, such definition of partial correctness can also be extended to general programs to smooth the correctness definition of programs so that language models can harvest more learning signal from the self-sampled programs. Building on top of the self-training framework, we propose NExT in Chapter 4, a method that teaches LLMs to reason about program execution. We introduce an LLM-friendly trace representation and augment the programs with such representation so that LLMs can generate natural language rationales that are grounded in these ground truth execution traces. By iterative training the LLMs to naturalize these execution traces into natural language rationales that leads to correct program outputs, we found that LLMs can generate higher quality rationales that can be used to directly communicate with the users and greatly boost the interpretability of LLMs for code-related applications. While these two methods improve LLMs with model tuning, in Chapter 5, we show how we can improve the language-to-code generation capabilities of LLMs without updating their parameters. We achieve this with Lever, by learning a smaller model to verify the LLM-generated code with its execution results. Through this work, we show that it is possible to inject execution semantics into the LLMs by adding smaller models as "add-ons" to rerank the program

samples. Finally in Chapter 6, we present `L2CEval`, a systematic evaluation of LLMs on 7 language-to-code generation tasks, across domains as semantic parsing, math reasoning and Python programming. In `L2CEval`, show that the language-to-code generation capabilities are strongly tied with the amount of training data, either during pre-training or instruction tuning stages. We also study the data contamination issue when evaluating LLMs for code generation in Chapter 6. By using surface-form and semantic matching methods, we quantify the overlapping percentage of popular code generation benchmarks and the open training data for language models on code.

7.1 Reflections

Here I share several lessons that I have learned and views that I have developed over the years of my dissertation research on program synthesis and language models for code.

Learning to execution vs. learning to use execution. In three works introduced in this dissertation, *i.e.,* Chapter 3, Chapter 4, Chapter 5, we use a program executor to provide execution information which helps improving the performance of language models for various program synthesis and code generation tasks. Different from these works, another direction for modeling program execution is by directly learning to predict the execution states, *i.e., learning to execution* (Nye et al., 2021; Zaremba and Sutskever, 2014), which aims to learn a model to simulate program execution. While learning to execute enjoys the benefit of virtually unlimited synthetic training data, it also requires strong reasoning capabilities that even the largest LLMs do not possess.[1] While LLMs are not yet able to simulate a program executor, I argue that they do not necessarily need to. Using an actual program executor is usually faster and more reliable, and the execution information it produces, let it be final execution results (*i.e.,* Chapter 5) or execution traces (*i.e.,* Chapter 3 and Chapter 4), can be used by the LLMs to better understand the program execution process and help the downstream tasks. The most significant limitation for learning to use execution lies in the cost of execution itself. With LLMs being able to generate more and

1. In the GPT-3 paper (Brown et al., 2020), they show that a 175B model can not reliably perform two digit multiplication.

more complex programs within a rather large code base, the procedure to systematically execute them might not be straightforward as the function-level program synthesis tasks we attempt to solve with most code generation benchmarks. Moreover, some programs might not always be safe to execute, yet we would still like to perform some runtime analysis on them. Human programmers are able to perform such "mental execution" and reason about program behavior at different levels of abstraction, which is a skill that current LLMs lack. Thus I believe learning to execute, or learning to reason about program execution **reliably**, is a key challenge that must be tackled on pursuing human-level programming capabilities for LLMs or AI in general.

Self-training for code. Another idea that is frequently used in this dissertation is self-training as the works in Chapter 3 and Chapter 4, which can also be seen as a way to create synthetic data. However, such self-training methods only find alternative solutions to the existing problems, *i.e.,* adding more pairs of $\{x_i, y_i^{'}\}$ while $\{x_i, y_i\}$ is an existing training example. While this method could teach the LLMs about program aliasing, *i.e.,* there are more ways to implement the same program, ultimately no new inputs (*i.e.,* x) are created in this self-training routine. While models get more capable, learning from self-sampled examples can still be a way to offset overfitting during finetuning, but it will be less powerful as the model already learns the polymorphism of programs. At that time, it would be crucial to develop methods that can create **pairs** of natural language descriptions and programs automatically to self-train the LLMs. In addition, self-training works well for code because the evaluation of code is relatively easy through execution, but many other language understanding or generation tasks do not enjoy such benefits. We will discuss more about this in § 7.2.

7.2 Future Work

While this dissertation targets on addressing several challenges for using large language models on the task of program synthesis from natural language, much more progress needs to be made for such LLM-based systems to be more capable and robust. Looking forward, I

plan to build upon my existing work and pursue the following directions.

Neuro-Symbolic approaches for reasoning. Code generation is not only a practical task that has real-world applications, it also resembles a different way for solving reasoning tasks using language models. Methods like chain-of-thought reasoning (Wei et al., 2022c; Zhou et al., 2022a) prompts the model not only to come up with the plan to solve the problem, but also requires precision in executing each steps that may require additional skills (*e.g.,* arithmetic computation). Recently, there has been a surge of generating and executing programs to approach reasoning tasks (Chen et al., 2022b; Gao et al., 2022), and LLMs only needs to translate the problems into program representations then take the output from the symbolic executor. This approach takes the best of two worlds, the versatility of language models and the precision and efficiency of symbolic systems. I plan to study how we can further integrate the neural language models and symbolic executors, more specifically on how to combine the autoregressive generation of LLMs with symbolic simulations or solvers to search for optimal completions, resonating with methods in recent work (Wang et al., 2024b; Ye et al., 2024).

Scaling self-training to more general tasks. In this dissertation, we introduce two self-training method to improve math reasoning and code execution reasoning abilities of LLMs. Such "sample→filter→train" methods work quite well for code and math tasks as the evaluating of such tasks are typically *automatic* and *objective*, thus we can easily filter out the low-quality model outputs and train the higher quality ones for self-improvement. However, other more general language understanding or generation tasks are much harder to evaluate, and there is a spectrum of such tasks. For example, the evaluation for tasks as commonsense reasoning and question answering is still objective and can be largely automatic, but for tasks as text summarization or creative story writing, the evaluation is much harder as it is quite subjective and often need human effort in evaluation. Thus it raises the question of how to apply self-improvement methods to these tasks that are harder to evaluate. I plan to explore the space of LLM-assisted evaluation and the multi-agent learning setup with LLMs as both the generator and the evaluator. It would also be interesting to transfer

and generalize the definition of partial correctness to general reasoning tasks under this framework.

Multi-modal modeling for code. In Chapter 4, we show that we can include the natural language description, the program and its execution traces in the prompt and have LLMs attending to all such inputs when generating outputs. These three types of inputs can also be regarded as three different *modalities* of the program. Besides these three textual modalities, code can often be represented in other modalities, such as graphs (*e.g.,* control graph, data flow graph, UML) and images (*e.g.,* websites, GUI, charts). More importantly, the conversion between these modalities are usually cheap and automatic. For example, one can simply use "`ast.parse()`" to obtain the abstract syntax tree (AST) (*i.e.,* a graph) from the textual surface form of a Python program. With the advancement of the large multi-modal models, it will be interesting to explore whether LLMs can achieve better understanding of the programs by jointly learning from different modalities. At the same time, since it is typically easy to obtain other modalities of the same program, there is also lots of potential in creating synthetic training data for training multi-modal models.

Beyond function-level code generation. While current evaluation of coding capabilities of LLMs almost exclusively focus on function-level code generation, popularized by benchmarks as MBPP (Austin et al., 2021a) and HumanEval (Chen et al., 2021a), code generation in the actual software development environment typically require the modeling of much longer context across different files, and the generated code must be grounded in such context. While repository-level code generation benchmarks exist (Shrivastava et al., 2023; Zhang et al., 2023), the performance of current LLMs on such benchmarks are quite disappointing, indicating a huge gap in generalizing from function-level code generation tasks to repository-level code generation tasks, and I seek to bridge this gap with my future research. Moreover, code generation itself is only one step of the software development process. There are plenty of other tasks need to be done for a human software engineer, such as program repair (*i.e.,* debugging), code review, software testing, code refactoring, code migration, etc. I also plan to do research beyond the scope of code generation, and

study how to build towards foundation models for automated software engineering.

Autonomous coding agents. When human programmers write programs, we interact with not only the code editor, but also look for documentations through the browsers, run commands and read information from command-lines, or even pair-program with colleagues. Recent work such as Devin [2], OpenDevin [3] and SWE-Agent (Yang et al., 2024) advance the field of automated software development by introduce autonomous coding agents that have their own browser and terminal to interact with. But results on benchmarks as SWE-bench (Jimenez et al., 2023) displays a clear gap between the skills of LLMs and those of human programmers on tasks as submitting pull requests to solve GitHub issues. I plan to make progress in building autonomous coding agents by collecting or synthesizing training data for LLMs that could potentially improve their ability to perform tasks as searching and reading documentations, interpreting the output from command-line, and reasoning about program behavior across different files in the repository.

2. `https://www.cognition.ai/blog/introducing-devin`

3. `https://opendevin.github.io/OpenDevin/`

Bibliography

Replit model. `https://huggingface.co/replit/replit-code-v1-3b`. Accessed: 2023-09-30.

Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28:793–819, 2019.

Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. Learning to generalize from sparse and underspecified rewards. In *International Conference on Machine Learning*, pages 130–140. PMLR, 2019.

Rajas Agashe, Srini Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *ArXiv*, abs/1910.02216, 2019a.

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446, Hong Kong, China, November 2019b. Association for Computational Linguistics. doi: 10.18653/v1/D19-1546. URL `https://aclanthology.org/D19-1546`.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of

code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Maitha Alhammadi, Mazzotta Daniele, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of language models: Towards open frontier models. 2023.

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, et al. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8: 556–571, 2020.

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(1):29–81, 1995.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. PaLM 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Thomas W. Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Neural Information Processing Systems (NeurIPS)*, 2017.

Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In *The Eleventh International Con-*

ference on Learning Representations, 2023. URL https://openreview.net/forum?id=Bo7eeXm6An8.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021a.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. *arXiv preprint arXiv:2304.01373*, 2023a.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward

Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. Pythia: A suite for analyzing large language models across training and scaling. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 2397–2430. PMLR, 23–29 Jul 2023b. URL `https://proceedings.mlr.press/v202/biderman23a.html`.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, October 2020.

David Bieber, Rishab Goel, Daniel Zheng, H. Larochelle, and Daniel Tarlow. Static prediction of runtime errors by learning to execute programs with external resource descriptions. *ArXiv*, abs/2203.03771, 2022.

Ekaba Bisong and Ekaba Bisong. Google bigquery. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pages 485–517, 2019.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable Forth interpreter. *ArXiv*, abs/1605.06640, 2016.

Islem Bouzenia, Yangruibo Ding, Kexin Pei, Baishakhi Ray, and Michael Pradel. TraceFixer: Execution trace-driven program repair. *ArXiv*, abs/2304.12743, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al.

Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, pages 1–17, 2023. doi: 10.1109/TSE.2023.3267446.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

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

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018a.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gfOiAqYm.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 22196–22208, 2021c. URL https://proceedings.neurips.cc/paper/2021/hash/ba3c95c2962d3aab2f6e667932daa3c5-Abstract.html.

Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34, 2021d.

Xinyun Chen, Dawn Xiaodong Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *ArXiv*, abs/2107.00101, 2021e.

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

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Monperrus Martin. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47:1943–1959, 2018b.

Jianpeng Cheng and Mirella Lapata. Weakly-supervised neural semantic parsing with a generative ranker. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*, pages 356–367, 2018.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*, 2022.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed Huai hsin Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *ArXiv*, abs/2210.11416, 2022.

J Cito, I Dillig, V Murali, and S Chandra. Counterfactual explanations for models of code. *International Conference on Software Engineering (ICSE)*, 2022.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Michael Collins and Terry Koo. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70, 2005.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. Free dolly: Introducing the world's first truly open instruction-tuned llm, 2023. URL `https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm`.

Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard Hovy. Iterative search for weakly supervised semantic parsing. 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 2669–2680, 2019a.

Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard H. Hovy. Iterative search for weakly supervised semantic parsing. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019b.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. Investigating data contamination in modern benchmarks for large language models. *arXiv preprint arXiv:2311.09783*, 2023.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. 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 4171–4186, Minneapolis, Minnesota, June 2019a. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019b. doi: 10.18653/v1/n19-1423. URL `https://doi.org/10.18653/v1/n19-1423`.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. doi: 10.18653/v1/p16-1004. URL `https://doi.org/10.18653/v1/p16-1004`.

Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K. Lahiri. Towards generating functionally correct code edits from natural language issue descriptions. *ArXiv*, abs/2304.03816, 2023.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481, 2022.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir R Radev. Improving text-to-sql evaluation methodology. In *ACL*, 2018.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Yao Fu, Hao Peng, and Tushar Khot. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*, 2022.

Yao Fu, Hao-Chun Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning (ICML)*, 2023.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning (ICML)*, 2016.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1809–1819, 2018.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *ArXiv*, abs/1410.5401, 2014.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, et al. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*, 2024.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*, 2024a.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024b.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45 (137):3–15, 2016.

Sumit Gulwani and Mark Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 803–814. ACM, 2014. doi: 10.1145/2588555.2612177. URL `https://doi.org/10.1145/2588555.2612177`.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio Cesar Teodoro Mendes, Allison Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero C. Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, S. Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuan-Fang Li. Textbooks are all you need. *ArXiv*, abs/2306.11644, 2023.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Xiaodong Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *ArXiv*, abs/2007.08095, 2020.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. DeepFix: fixing common C language errors by deep learning. In *AAAI Conference on Artificial Intelligence*, 2017.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Association for Computational Linguistics (ACL)*, 2017a.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *ACL*, 2017b.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, et al. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2020. URL `https://arxiv.org/abs/2006.03654`.

Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. Synthesizing research on programmers' mental models of programs, tasks and concepts - a systematic literature review. *Inf. Softw. Technol.*, 164:107300, 2022.

Charles T. Hemphill, John J. Godfrey, and George R. Doddington. The ATIS spoken language systems pilot corpus. In *Speech and Natural Language: Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27,1990*, 1990. URL `https://aclanthology.org/H90-1021`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Danny Hernandez, Tom Brown, Tom Conerly, Nova DasSarma, Dawn Drain, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Tom Henighan, Tristan Hume, et al. Scaling laws and interpretability of learning from repeated data. *arXiv preprint arXiv:2205.10487*, 2022.

Matthew Douglas Hoffman, Du Phan, David Dohan, Sholto Douglas, Tuan Anh Le, Aaron T Parisi, Pavel Sountsov, Charles Sutton, Sharad Vikram, and Rif A Saurous. Training Chain-of-Thought via Latent-Variable inference. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Xing Hu, Ge Li, Xin Xia, D. Lo, and Zhi Jin. Deep code comment generation. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010, 2018.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *arXiv preprint arXiv:2207.05608*, 2022.

Andy Hunt and David Thomas. The pragmatic programmer: From journeyman to master. 1999.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436, 2019.

Srini Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *ArXiv*, abs/1808.09588, 2018.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*, 2016.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023a.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024a.

Minhao Jiang, Ken Ziyu Liu, Ming Zhong, Rylan Schaeffer, Siru Ouyang, Jiawei Han, and Sanmi Koyejo. Investigating data contamination for pre-training language models. *arXiv preprint arXiv:2401.06059*, 2024b.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442, 2023b.

Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. Omnitab: Pretraining with natural and synthetic data for few-shot table-based question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 932–942, 2022.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Tim Johnson. Natural language computing: the commercial applications. *The Knowledge Engineering Review*, 1(3):11–23, 1984.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El-Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, Deep Ganguli, Danny Hernandez, Josh Jacobson, Jackson Kernion, Shauna Kravec, Liane Lovitt, Kamal Ndousse, Catherine Olsson, Sam Ringer, Dario Amodei, Tom Brown, Jack Clark, Nicholas Joseph, Ben Mann, Sam McCandlish, Chris Olah, and Jared Kaplan. Language models (mostly) know what they know, 2022. URL `https://arxiv.org/abs/2207.05221`.

Sungmin Kang, B. Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *ArXiv*, abs/2304.02195, 2023.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Jens Krinke and Chaiyong Ragkhitwetsagul. Bigclonebench considered harmful for machine learning. *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7, 2022.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, 2017.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. SPoC: Search-based pseudocode to code. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *arXiv preprint arXiv:2211.11501*, 2022.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2023.

Jooyoung Lee, Thai Le, Jinghui Chen, and Dongwon Lee. Do language models plagiarize? *Proceedings of the ACM Web Conference 2023*, 2022. URL `https://api.semanticscholar.org/CorpusID:247450984`.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965. URL `https://api.semanticscholar.org/CorpusID:60827152`.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yi Li, Shaohua Wang, and Tien Nhut Nguyen. DLFix: Context-based code transformation

learning for automated program repair. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 602–614, 2020.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. On the advance of making language models better reasoners. *arXiv preprint arXiv:2206.02336*, 2022a.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022c.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022d.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *ACL*, pages 23–33, 2017.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31, 2018.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee,

Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03*, 1, 2017a.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2017b.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.*, 2018.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. Bridging textual and tabular data for cross-domain text-to-sql semantic parsing. *arXiv preprint arXiv:2012.12627*, 2020.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. Tapex: Table pre-training via learning a neural sql executor. In *International Conference on Learning Representations*, 2021.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized

BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL `http://arxiv.org/abs/1907.11692`.

S. Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning (ICML)*, 2023.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021a.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664, 2021b.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023.

Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Yefei Liu, Cen Zhang, Liming Nie, and Yang Liu. ChatGPT: Understanding code syntax and semantics. 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *ArXiv*, abs/2303.17651, 2023.

Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning*, 38(4):1046–1061, 2022.

Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. A discrete hard em approach for weakly supervised question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2851–2864, 2019.

Swaroop Mishra, Matthew Finlayson, Pan Lu, Leonard Tang, Sean Welleck, Chitta Baral, Tanmay Rajpurohit, Oyvind Tafjord, Ashish Sabharwal, Peter Clark, et al. Lila: A unified benchmark for mathematical reasoning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5807–5832, 2022.

Dipendra Misra, Ming-Wei Chang, Xiaodong He, and Wen-tau Yih. Policy shaping and generalized update equations for semantic parsing from denotations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2442–2452, 2018.

Arindam Mitra, Luciano Del Corro, Shweti Mahajan, Andres Codas, Clarisse Simoes, Sahaj Agrawal, Xuxi Chen, Anastasia Razdaibiedina, Erik Jones, Kriti Aggarwal, Hamid Palangi, Guoqing Zheng, Corby Rosset, Hamed Khanpour, and Ahmed Awadallah. Orca 2: Teaching small language models how to reason. *ArXiv*, abs/2311.11045, 2023.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. OctoPack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Hassan Awadallah. Orca: Progressive learning from complex explanation traces of GPT-4. *ArXiv*, abs/2306.02707, 2023.

Mahdi Pakdaman Naeini, Gregory Cooper, and Milos Hauskrecht. Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

Ansong Ni, Pengcheng Yin, and Graham Neubig. Merging weak and active supervision for semantic parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8536–8543, 2020.

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Oleksandr Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. Learning from self-sampled correct and partially-correct programs. *arXiv preprint arXiv:2205.14318*, 2022.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning (ICML)*, pages 26106–26128. PMLR, 2023a.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. L2ceval: Evaluating language-to-code generation capabilities of large language models. *arXiv preprint arXiv:2309.17446*, 2023b.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.

Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations*, 2020a.

Maxwell Nye, Armando Solar-Lezama, Josh Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *Advances in Neural Information Processing Systems*, 33:10832–10842, 2020b.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Demystifying GPT self-repair for code generation. *arXiv preprint arXiv:2306.09896*, 2023.

OpenAI. Chatgpt: Optimizing language models for dialogue, November 2022. URL `https://openai.com/blog/chatgpt/`.

OpenAI. GPT-4 technical report, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1142. URL https://aclanthology.org/P15-1142.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*, 2021.

Rishov Paul, Md. Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna C. S. Santos. Enhancing automated program repair through fine-tuning and prompt engineering. 2023.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning (ICML)*, 2023.

Zhen Peng, Zhizhi Wang, and Dong Deng. Near-duplicate sequence search at scale for large language model memorization evaluation. *Proceedings of the ACM on Management of Data*, 1:1 – 18, 2023. URL https://api.semanticscholar.org/CorpusID:259213212.

Archiki Prasad, Swarnadeep Saha, Xiang Zhou, and Mohit Bansal. Receval: Evaluating reasoning chains via correctness and informativeness. *arXiv preprint arXiv:2304.10703*, 2023.

Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. Rasat: Integrating relational structures into pretrained seq2seq model for text-to-sql. *arXiv preprint arXiv:2205.06983*, 2022.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149,

Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1105. URL https://aclanthology.org/P17-1105.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

Nazneen Fatema Rajani, Bryan McCann, Caiming Xiong, and Richard Socher. Explain yourself! leveraging language models for commonsense reasoning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 4932–4942, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1487.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*, 2022.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI 2015*, 2015.

Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. Impact of pretraining term frequencies on few-shot numerical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 840–854, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.59. URL https://aclanthology.org/2022.findings-emnlp.59.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*, 2024.

Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael J. Muller, and Justin D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. JUNITMZ at SemEval-2016 task 1: Identifying semantic similarity using Levenshtein ratio. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 702–705, San Diego, California, June 2016. Association for Computational Linguistics. doi: 10.18653/v1/S16-1108. URL `https://aclanthology.org/S16-1108`.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, 2021.

Tal Schuster, Ashwin Kalyan, Alex Polozov, and Adam Tauman Kalai. Programming puzzles. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

Priyanka Sen and Emine Yilmaz. Uncertainty and traffic-aware active learning for semantic parsing. In *Proceedings of the First Workshop on Interactive and Executable Semantic Parsing*, pages 12–17, 2020.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2269–2279, 2021.

Libin Shen, Anoop Sarkar, and Franz Josef Och. Discriminative reranking for machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, pages 177–184, 2004.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*, 2022a.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. CrossBeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations (ICLR)*, 2022b.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. ExeDec: Execution decomposition for compositional generalization in neural program synthesis. In *International Conference on Learning Representations (ICLR)*, 2024.

Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. On the potential of lexico-logical alignments for semantic parsing to SQL queries. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1849–1864, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.167. URL `https://aclanthology.org/2020.findings-emnlp.167`.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31. Curran Associates, Inc., 2018.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting

grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10737–10746. IEEE, 2020.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR, 2023.

Vered Shwartz, Peter West, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Unsupervised commonsense question answering with self-talk. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4615–4629, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.373.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 269–274, 2014. doi: 10.1109/ISSREW.2014.36.

Riley Simmons-Edler, Anders Miltner, and Sebastian Seung. Program synthesis through reinforcement learning guided tree search. *arXiv preprint arXiv:1806.02932*, 2018.

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. Nofuneval: Funny how code lms falter on requirements beyond functional correctness. *arXiv preprint arXiv:2401.15963*, 2024.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of ChatGPT. *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30, 2023.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming

by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, 2005.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

Phillip D Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023a. Accessed: 2023-05-05.

MosaicML NLP Team. Introducing mpt-30b: Raising the bar for open-source foundation models, 2023b. Accessed: 2023-06-22.

Marjorie Templeton and John D. Burger. Problems in natural-language interface to DSMS with examples from EUFID. In *1st Applied Natural Language Processing Conference, ANLP 1983, Miramar-Sheraton Hotel, Santa Monica, California, USA, February 1-3, 1983*, pages 3–16. ACL, 1983. doi: 10.3115/974194.974197. URL `https://aclanthology.org/A83-1002/`.

Kushal Tirumala, Daniel Simig, Armen Aghajanyan, and Ari Morcos. D4: Improving llm pretraining via document de-duplication and diversification. *Advances in Neural Information Processing Systems*, 36, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, L. Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *ArXiv*, abs/2211.14275, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Richard J Waldinger and Richard CT Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252, 1969.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, 2020.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`, May 2021.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-SQL generation with execution-guided decoding. *arXiv: Computation and Language*, 2018a.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding, 2018b. URL `https://arxiv.org/abs/1807.03100`.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100*, 2018c.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):1–26, 2024a.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *ArXiv*, abs/2312.08935, 2024b.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. *arXiv preprint arXiv:2205.02029*, 2022a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022b.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685`.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages. In *Findings*, 2022c.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*, 2022d.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2021.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani

Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022a.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *"arXiv preprint arXiv:2201.11903"*, January 2022b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022c.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*, 2022.

William A. Woods. Progress in natural language understanding: an application to lunar geology. In *American Federation of Information Processing Societies: 1973 National Computer Conference, 4-8 June 1973, New York, NY, USA*, volume 42 of *AFIPS Conference Proceedings*, pages 441–450. AFIPS Press/ACM, 1973a. doi: 10.1145/1499586.1499695. URL `https://doi.org/10.1145/1499586.1499695`.

William A Woods. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 441–450, 1973b.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Chun Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

Chun Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *ArXiv*, abs/2304.00385, 2023.

Chun Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, 2023.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1341–1350, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1127. URL `https://aclanthology.org/P16-1127`.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I Wang, et al. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *arXiv preprint arXiv:2201.05966*, 2022.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Deep Learning for Code Workshop*, 2022.

Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, Zhoujun Cheng, Siheng Zhao, Lingpeng Kong, Bailin Wang, Caiming Xiong, and Tao Yu. Lemur: Harmonizing natural language and code for language agents, 2023.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Monperrus Martin. SelfAPR: Self-supervised program repair with test execution diagnostics. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36, 2024.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL `https://aclanthology.org/P17-1041`.

Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*, 2018.

Pengcheng Yin and Graham Neubig. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1447. URL `https://aclanthology.org/P19-1447`.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486, 2018a.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486, 2018b.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*, 2020.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 126–173, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.9.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*, 2018a.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium, October-November 2018b. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL `https://aclanthology.org/D18-1425`.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. Grappa: Grammar-augmented pre-training for table semantic parsing. *arXiv preprint arXiv:2009.13845*, 2020.

Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. CodeIPPrompt: Intellectual property infringement assessment of code language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 40373–40389. PMLR, 23–29 Jul 2023. URL `https://proceedings.mlr.press/v202/yu23g.html`.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *ArXiv*, abs/1410.4615, 2014.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems (NeurIPS)*, 35: 15476–15488, 2022.

John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pages 1050–1055. AAAI Press / The MIT Press, 1996a. URL `http://www.aaai.org/Library/AAAI/1996/aaai96-156.php`.

John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996b.

Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, pages 658–666. AUAI Press, 2005. URL `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1209&proceeding_id=21`.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

Tianyi Zhang, Tao Yu, Tatsunori B Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I Wang. Coder reviewer reranking for code generation. *arXiv preprint arXiv:2211.16490*, 2022.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shanshan Wang, Yufei Xue, Zi-Yuan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *ArXiv*, abs/2303.17568, 2023.

Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-sql with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411, 2020.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017a. URL `http://arxiv.org/abs/1709.00103`.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017b.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022a.

Fan Zhou, Mengkang Hu, Haoyu Dong, Zhoujun Cheng, Shi Han, and Dongmei Zhang. Tacube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data. *arXiv preprint arXiv:2205.12682*, 2022b.

Shuyan Zhou, Pengcheng Yin, and Graham Neubig. Hierarchical control of situated agents through natural language. *arXiv preprint arXiv:2109.08214*, 2021.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*, 2023.