Can LLMs Implicitly Learn Numeric Parameter Constraints in Data Science APIs?

Yinlin Deng **Chunqiu Steven Xia** **Zhezhen Cao** **Meiziniu Li** **Lingming Zhang**

University of Illinois Urbana-Champaign
Southern University of Science and Technology
The Hong Kong University of Science and Technology

{yinlind2,chunqiu2,lingming}@illinois.edu, 12110529@mail.sustech.edu.cn, mlick@cse.ust.hk

Abstract

Data science (DS) programs, typically built on popular DS libraries (such as PyTorch and NumPy) with thousands of APIs, serve as the cornerstone for various mission-critical domains such as financial systems, autonomous driving software, and coding assistants. Recently, large language models (LLMs) have been widely applied to generate DS programs across diverse scenarios, such as assisting users for DS programming or detecting critical vulnerabilities in DS frameworks. Such applications have all operated under the assumption, that LLMs can implicitly model the numerical parameter constraints in DS library APIs and produce valid code. However, this assumption has not been rigorously studied in the literature. In this paper, we empirically investigate the proficiency of LLMs to handle these implicit numerical constraints when generating DS programs. We studied 28 widely used APIs from PyTorch and NumPy, and scrutinized the LLMs' generation performance in different levels of granularity: full programs, all parameters, and individual parameters of a single API. We evaluated both state-of-the-art open-source and closed-source models. The results show that LLMs are great at generating simple DS programs, particularly those that follow common patterns seen in training data. However, as we increase the difficulty by providing more complex/unusual inputs, the performance of LLMs drops significantly. We also observe that GPT-4-Turbo can sustain much higher performance overall, but still cannot handle arithmetic API constraints well. In summary, while LLMs exhibit the ability to memorize common patterns of popular DS API usage through massive training, they overall lack genuine comprehension of the underlying numerical constraints.

1 Introduction

Data science (DS) is an emerging and important area that combines classic fields like statistics, databases, data mining, and machine learning (ML) to gain insights via complex operations on the abundance of available data [49]. DS libraries (such as PyTorch [41] and NumPy [38]) contain thousands of APIs used by developers and data scientists to process/analyse data. These DS APIs serve as the fundamental building blocks for almost all important ML/DS pipelines, and have penetrated into almost every corner of modern society, including financial systems [18, 4], autonomous driving software [9, 27, 46], coding assistants [45, 37], etc. Due to their high importance and wide usage, automatically synthesizing valid DS programs has been a critical research area [29, 21, 47].

One key challenge of DS code generation is to satisfy the complex constraints within each DS library API. DS library APIs perform transformations (e.g., matrix multiplication) on inputs (i.e., arrays or array-like objects) with numeric constraints on API parameters and inputs. Figure 1 shows an example

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

of a typical *DS program* where the DS library (i.e., PyTorch) is first imported, followed by creating some `input_data`, and then performing the data manipulation operation on the `input_data` using a DS API (`torch.nn.Conv2d`). The parameters of the API (e.g., `kernel_size`, `groups`) must satisfy the corresponding constraints between API parameters and the properties of the `input_data`. We refer to *API constraints* as the set of relationships between properties of `input_data` and API parameters that, if and only if when satisfied, leads to a valid DS API invocation. As seen in Figure 1, not only are there constraints between the properties of the `input_data` and API parameters (e.g., `kernel_size` $\leq$ `H + 2*padding`), but there are also constraints within API parameters (e.g., `out_channel % groups = 0`). These constraints are defined by developers according to the functionality of each DS API, and are usually specified in natural language within the API documentation. Such complex constraints are critical for DS applications, and DS users or even DS experts may unintentionally violate such constraints [29, 26].

Large language models (LLMs) have achieved tremendous success in processing code [10, 2]. Due to their powerful code understanding and generation ability, LLMs have been applied to various coding tasks [34], such as code completion [20, 6], program repair [19, 54], and test generation [16, 17, 48]. For DS libraries, LLMs have been applied to solve practical user queries on StackOverflow [29] and even generate test programs to detect bugs in modern ML frameworks [16]. Prior work assumes LLMs, through massive training, can already implicitly model constraints in DS APIs by learning from numerous correct DS API uses [47, 21, 16]. However, this assumption has not been systematically verified. Furthermore, popular DS-specific benchmarks like DS-1000 [29] do not specially test the LLM's ability to satisfy implicit constraints and instead focus on how to apply DS APIs to solve data analysis tasks. These gaps in prior research raise a critical question: *Can LLMs implicitly learn the numeric constraints in data science APIs?*

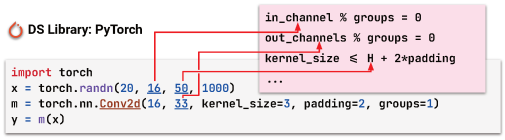


Figure 1: Example DS program with constraints

Our work. To answer the question, we conduct a rigorous study on the performance of LLMs in generating valid DS programs satisfying diverse numerical API constraints. We collected a set of 28 representative DS library APIs across two widely-used Python DS libraries (PyTorch and NumPy), each with their unique constraints/setup. Additionally, we categorize each API's constraints into different categories (e.g., equality and arithmetic) and perform in-depth experiments on each constraint type. To support our analysis, we systematically created 3 generation settings: full program, all parameters, and individual parameters, designed to test the LLMs under different evaluation scenarios. Additionally, we vary the difficulty level by adjusting the inputs to explore LLM behaviours when asked to solve more complex API constraints or given more unnatural inputs.

Interestingly, contrary to the popular assumption in prior work, while LLMs can easily satisfy constraints when the inputs are simple, we observe that the performance drops drastically as we increase the difficulty or provide more unusual inputs. We found that LLMs tend to generate simple and common inputs seen during training, highlighting that LLMs are often memorizing patterns instead of truly understanding the actual DS API constraints. For example, for the widely used `Conv2d` API shown in Figure 1, when `max(in_channels,out_channels)` is set to `[128, 256)`, even GPT-4-Turbo [1] can only predict the correct value of `groups` $\sim$24% of the time, while the other models are below 14%. Furthermore, based on our experimental findings, we constructed DSEVAL, the first benchmark for systematically evaluating LLMs' capabilities in understanding the important numerical API constraints for popular DS libraries. DSEVAL contains 19,600 different problems across 12 representative APIs to extensively compare and contrast the performance of different LLMs. DSEVAL supports lightweight and fast evaluation by extracting LLM generated parameters and quickly verifying the correctness using state-of-the-art SMT solvers (such as Z3 [13]) to avoid time-consuming execution-based evaluations. Our evaluation on eight state-of-the-art open-source and closed-source models shows that while all studied models struggle with more difficult problems, GPT-4-Turbo consistently achieves the highest accuracy across all difficulty levels. For example, GPT-4-Turbo achieves an average accuracy of 57.5% for *hard* constraints of PyTorch APIs, while the best open-source model can only achieve 39.2%, demonstrating the huge gap between large proprietary models and other open-source LLMs. Our design of DSEVAL is general and can be easily extended to additional libraries and APIs for the DS domain and even beyond.

Table 1: Categorization of constraint types with exemplar API names, description, and examples.

Category	API names	Description	Example
Equality	`BatchNorm2d`, `Linear` `squeeze`, `split`	Copying specific dimension Indexing the correct dimension	`nfeat = input_shapes[1]` `input_shapes[axis] = 1`
Inequality	`SoftMax`, `mean` `sum`, `max`	Single value related to rank Multiple values related to rank	`-rank ≤ dim < rank` `-rank ≤ dim < rank for dim in dims`

Arithmetic	`MaxPool2d`, `AvgPool2d`
	`Conv2d`, `Conv1d`
	`reshape`, `reshape`
	`Fold`, `Conv1d`
		Multiplies a constant number
		Divides a parameter
		Product of parameters
		Complex arithmetic
			`kernel_size ≤ H + padding * 2`
			`in_channels % groups = 0`
			$\prod$`input_shapes = `$\prod$`target_shape`
			$L=\prod\lfloor\frac{\texttt{o\_size[d]+2xpad[d]-dil[d]x(k\_size[d]-1)-1}}{\texttt{stride[d]}}+1\rfloor$

Set-related	`max`, `sum` `transpose`	Uniqueness Completeness	`|{dims}| = |dims|` `{input_shapes} = {axes}`

2 Study Approach

2.1 Scope of study

Instead of considering all possible DS programs and APIs, we focus on simple DS programs with only a single API call. This allows us to isolate the evaluation to individual APIs or even individual API parameters, facilitating fine-grained analysis and a detailed examination of the LLMs' limitations with respect to various types of numerical constraint.

We specifically target the core APIs commonly used by users that perform operations on the `input_data`. Additionally, we also only consider *numeric* API constraints: constraints with only numeric parameters such as integers. We ignore any other types of parameters (e.g., string) since they do not affect the validity of numeric constraints. As such, any non-numeric parameters produced by the model will be discarded during constraint validation.

Table 1 shows the types of constraints we considered in the study with the corresponding categories. We group the constraints into *i)* Equality: constraints where the values have to match exactly. We see that equality constraints are related to selecting or generating the right shape in the `input_data`. *ii)* Inequality: constraints where values have to be greater or less than. Inequality constraints include mainly rank related operations to stay within the valid rank range. *iii)* Arithmetic: constraints involving arithmetic operations such as division, modulus or products. There are also more complex API constraints that includes combination of many arithmetic operations. *iv)* Set-related: constraints where the satisfaction criteria depend on different set-based properties. For example, there are constraints that require parameters to be unique or complete with respect to `input_shapes`.

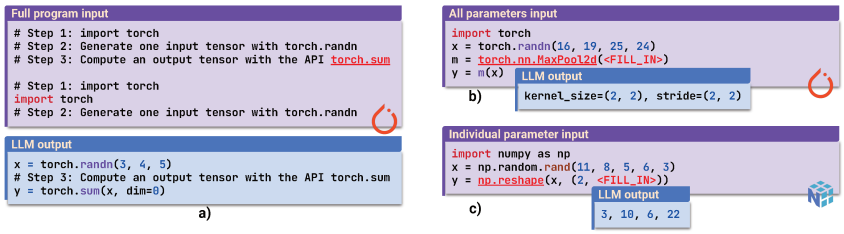


Figure 2: Example problem input and LLM output for each evaluation setting

2.2 Evaluation settings

Next, we describe our settings to evaluate the performance of LLM on handling the numeric constraints. In total, we have 3 settings: *i)* full program, *ii)* all parameters, and *iii)* individual parameter.

Full program. For the full program setting, we want the LLM to synthesize a complete DS program using a specific API from scratch. To do this, we provide a 3-step instruction and the basic starting code of importing the DS library. Figure 2a shows an example of the full program input for the API `torch.sum` as well as an example LLM output. We note that in this setting, the LLM has full

freedom to generate any type or size for the `input_data`. As such, the LLM may choose very simple `input_data` and API parameter values that can easily satisfy the constraint.

All parameters. In the all parameters setting, we directly provide the `input_data` for the API. Figure 2b also shows an example of the input for the API `torch.nn.MaxPool2d` where the LLM just needs to output the API parameters. This setting evaluates if/how LLMs can accurately solve the constraints as we vary the `input_data` with more difficult or uncommon cases. Still the LLM has full freedom to pick the full combination of parameters to satisfy the required constraint.

Individual parameter (main setting). To perform a finer-grained evaluation, we introduce the individual parameter setting where we ask the LLM to generate a single parameter of the API. Figure 2c additionally demonstrates an example for `np.reshape` where we only allow the LLM to fill in a single parameter value of `newshape`. Furthermore, we can also add an additional constraint by directly providing the first value of `newshape` (`2` in the example). This makes the problem even more challenging where instead of being able to simply copy the `input_shapes`, the LLM now has to reason with the partial shape given and compute the final correct shape to satisfy the constraint. Compared to the prior two settings, the choices here are much limited. This makes the task harder to fully evaluate how LLMs solve complex API constraints, and serves as our main setting.

2.3 Input creation and output validation

Creation. To produce the inputs for each of the 3 settings, we use a fixed set of templates for each API. For the full program setting, we produce one input per API, changing only the API name in the input instruction. For the all parameters setting, we vary the `input_data` given to the API. In particular, we focus on two properties of the `input_data`: 1) rank of the `input_data` and 2) each dimension value. We create randomized inputs and increase the difficulty by either increasing the rank or the dimension values to measure the LLM performance. Note that input rank or dimensionality can affect different APIs depending on the specific numeric constraints (Table 1). For example, an API like `torch.nn.SoftMax` that has a constraint of `-rank` $\leq$ `dim` $<$ `rank` will have its difficulty influenced by the actual rank of the input tensor. On the other hand, an API like `torch.nn.Conv2d` has a constraint of `in_channels % groups = 0`, which depends on the actual dimension value of the input (i.e., `in_channels`). As the dimension value of `in_channels` increases, it will be more difficult to select the `groups` parameter that can divide it evenly. Therefore, we increase the difficulty of different APIs based on whether the constraint depends on the rank, dimension, or both. Similarly, for the individual parameter setting, we also randomize the `input_data` based on the previous two properties. Additionally, we pick the parameters with interesting constraints for the LLM to predict in order to be representative and cover the major constraint types. Furthermore, since we only ask the LLM to produce a single parameter value, we also vary the other parameter values in the API to add additional constraints (details discussed in Section 4.3).

To ensure the input is valid, we leverage satisfiability modulo theory (SMT) solvers as shown in Figure 3a. SMT solvers, such as Z3 [13], are tools which can be used to solve an SMT problem of determining whether a mathematical or first-order logic formula is satisfiable [5]. We first encode the API constraints into an SMT formula. We then randomly generate *concrete values* for the `input_shapes` and leave the other parameters that we want the LLM to generate as *symbolic variables*. Next, we use an SMT solver to check if the constraints are satisfiable (i.e., there exists a set of values for each symbolic variable that can satisfy the constraint). If it is satisfiable, the input we provide to the LLM is valid, otherwise we restart the process by randomly selecting the concrete values. In our study, we reuse the encoded API constraints provided by NNSmith [32] (a popular tool for testing ML libraries via formal constraint solving) and add additional ones when needed.

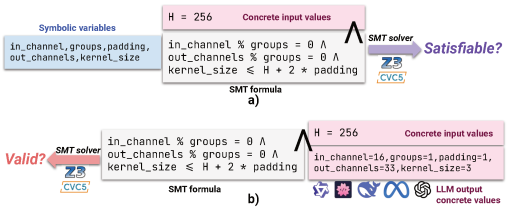


Figure 3: Example usage of constraint solvers to generate inputs and validate outputs.

Evaluation. To evaluate the validity of the DS programs generated by the LLMs, we first parse the output to extract the `input_data` and API parameters. We then check if the LLM predicted

values are valid. This is also done via SMT solving as demonstrated in Figure 3b where we use the SMT formulas and, this time, check if all the concrete values generated are valid according to the constraints. Note that such light-weight constraint solving can support much faster validation than actually executing the generated DS programs, while still providing the same guarantee.

3 Experimental Setup

3.1 Subjects

We construct a dataset with 28 representative APIs in total from two popular DS libraries: PyTorch (18) and NumPy (10). For our API selection process, we begin by referencing prior work NNSmith [32] and examined all 73 core operators it supports. From these, we select 22 core APIs that have numeric parameter constraints and add additional 6 APIs to obtain the 28 APIs used in our study for both the full program prediction setting (Section 4.1) and the full API parameter prediction setting (Section 4.2). For a more detailed analysis, we select 12 APIs to cover the representative types of numeric constraint for examination in the single API parameter prediction setting (Section 4.3) and in our DSEVAL benchmark (Section 4.4). We use "representative" to mean representative with respect to the numeric parameter constraints in DS library APIs. Table 1 shows the categorization of the different types of numeric constraints that exist in DS libraries. Our selection criteria aim to select a list of APIs that have interesting numeric parameter constraints that can cover all the major constraint categories. A complete list of the 12 APIs and their corresponding constraints is provided in Table 3 in the Appendix.

We focus on the 3 settings described previously to analyse the performance of LLMs. For the full program setting, we generate a single input prompt per each studied API and ask the LLMs to synthesize the complete DS program by varying the sampling temperature. For the all parameters setting, we have 14 difficulty settings, each with 200 different inputs per API, and use greedy decoding to obtain the LLM solutions. The difficulty setting is controlled by increasing the rank of `input_data` (from 2 to 8 in intervals of 1) with default dimension value as `[1,16]`, and increasing the dimension value (i.e., `[1,4)`, `[4,8)`,... , `[128,256])` with default rank as 3, separately. Finally, in the single parameter setting, we select one parameter for each API for the LLM to generate. For any parameters irrelevant to the constraint, we use the default value if it is an optional parameter, and randomly choose from a reasonable value range if it is a required parameter (Appendix C). We adopt the same difficulty setup and greedy decoding strategy as the all parameter setting.

3.2 Metrics

Validity. To measure validity, we directly extract the LLM output predictions and evaluate according to the process described in Section 2.3. We define *accuracy* as the percentage of valid programs produced by the LLMs in each difficulty setting.

Diversity. To measure diversity, we compute the *unique valid rate*: the percentage of unique valid programs generated via sampling. Note that we deduplicate by extracting the input shapes and numeric parameters, ignoring the irrelevant parameters and irrelevant code suffix.

3.3 Studied models.

We evaluate 8 popular state-of-the-art LLMs, including both closed-source and open-source models (detailed list shown in Table 2). For both the full program and all parameter settings, we only present the results for DeepSeek Coder-33b [22], state-of-the-art open-source model, due to the space limit (other models follow similar trends). For the individual parameter setting (the main setting), we focus on the DeepSeek Coder family models (33b, 6.7b, and 1.3b) as well as GPT-4-Turbo (2024-04-09), covering both state-of-the-art open-source and close-source models, as well as models with different sizes. Apart from the full program setting, where the LLM generates a complete program, we perform infilling using the studied LLMs' model-specific infilling format. To perform infilling using GPT-4-Turbo, we design a specialized prompt (see Appendix H). Unless otherwise stated, we use greedy decoding (i.e., temperature = 0) and temperature of 1 when sampling for diversity evaluation.

4 Evaluation

4.1 Full program prediction

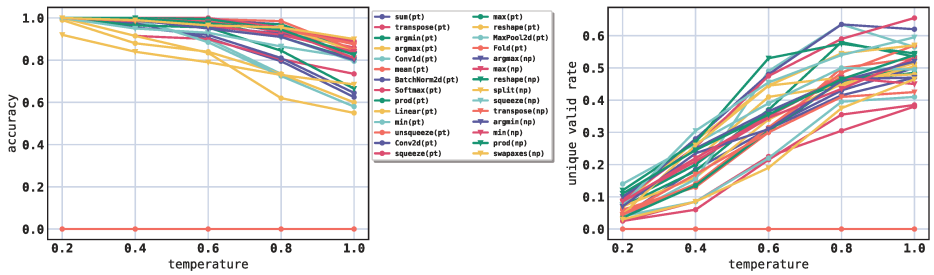


Figure 4: Full program prediction result on all 28 APIs (PyTorch and NumPy).

To start with, we ask the LLM (DeepSeek Coder-33b) to predict the entire DS program from scratch given just simple instructions. Figure 4a shows the overall accuracy of the 18 APIs in PyTorch and 10 APIs in NumPy. We see that with low temperature the model has near perfect accuracy on almost all the APIs and as temperature slowly increases, the accuracy tends to drop (ending with around 0.5∼0.8 with temperature=1). Surprisingly, we found that for `torch.nn.Fold`, which contains the most complex constraint, the LLM failed to produce any valid DS programs. This demonstrates that LLMs may still struggle with satisfying the extremely difficult constraints even when given the full freedom of generating any input values. Furthermore, in Figure 4b, we plot the proportion of unique valid programs generated by the model as we vary temperature. Of course when sampling at low temperatures, many of the inputs will be repeated, leading to low number of unique programs in general. In particular, the input shapes are often from widely-used computer vision datasets like `3*224*224` from ImageNet [15]. This indicates the LLMs tend to memorize some common patterns from either documentation or user programs. However, we see that even though the unique valid rate increases with high temperatures to give more diverse and creative outputs, the percentage of unique valid programs can still be mostly below 50%. This demonstrates that while models are successful in generating a high percentage of valid programs, a lot of generated programs are repeated.

4.2 Full API parameter prediction

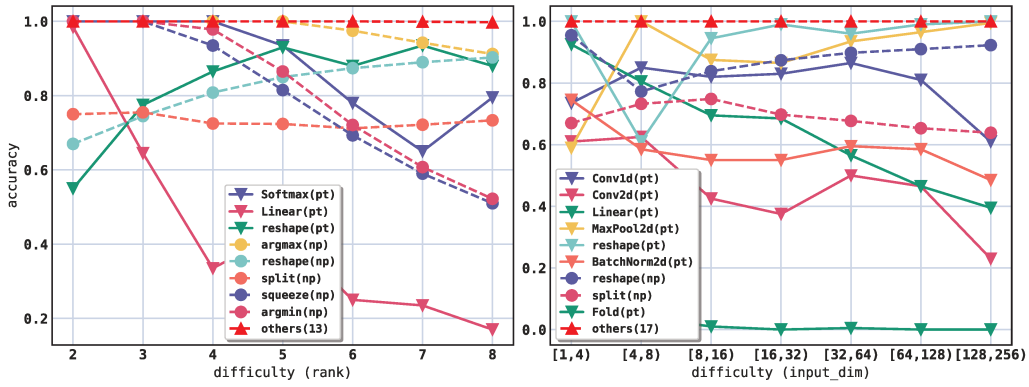


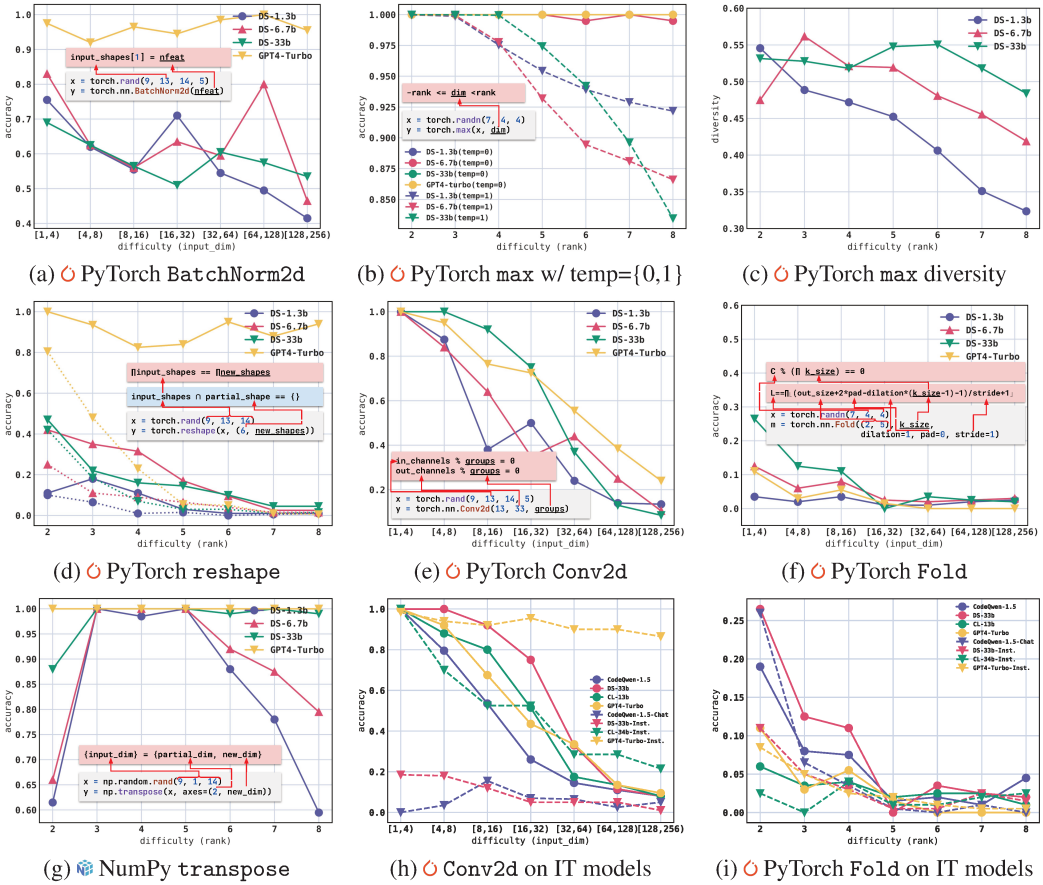
Figure 5: Full API parameter prediction result on all 28 APIs (PyTorch and NumPy). The LLM has near 100% accuracy on some APIs, which are collectively referred to as `others(x)`, where `x` is the number of grouped APIs.

Figure 5 shows the setting where we randomly provide an `input_data` and ask DeepSeek Coder-33b to complete the valid parameters of the API. We vary the difficulty by changing either the rank or the dimension value ranges of the `input_data` to produce more complex and unnatural inputs. We use greedy decoding (temperature 0) to generate one solution per problem, and compute the

average valid rate across the randomly created problems to compute accuracy for each difficulty level. Compared to Section 4.1 where LLMs achieve near-perfect accuracy for almost all APIs with low temperature like 0.2, we observe that the accuracy quickly drops when simply randomizing the input shape, especially for APIs with more complex constraints. This indicates that the learned patterns cannot easily generalize to less common input shapes. We further performed an interesting case study on the PyTorch API `Linear`, and found this phenomenon holds true across different models (Appendix D). However, we see that the majority of APIs maintain high accuracy even as difficulty increases (`others(x)` in Figure 5). This is because these APIs have relatively easy constraints. For example, APIs like `max` or `argmax` only require predicting a single integer representing the dimension to operate on, and the LLMs learn to predict `dim=1` or just rely on the default parameter values of the API which are always valid.

4.3 Single API parameter prediction

We now focus on the main finer-grained evaluation setting where we ask LLMs to predict a single parameter value and discuss the input setup, results, and findings for each API separately. Here, we only discuss representative API constraints from each category and full results are in Appendix F.



(a) PyTorch `BatchNorm2d` (b) PyTorch `max` w/ temp={0,1} (c) PyTorch `max` diversity

(d) PyTorch `reshape` (e) PyTorch `Conv2d` (f) PyTorch `Fold`

(g) NumPy `transpose` (h) `Conv2d` on IT models (i) PyTorch `Fold` on IT models

Figure 6: Single API parameter result. Solid lines (except Fig. 6c) show the accuracy of using greedy decoding (`temp=0`). In Fig. 6b, dashed lines show the pass@1 accuracy in sampling experiments with `temp=1`. In Fig. 6d, dotted lines show the accuracy after excluding trivial solutions. In Fig. 6h and 6i, we use *-Inst. to distinguish between the generation settings: infilling (GPT4-Turbo) and free-form generation (GPT4-Turbo-Inst.). More details are provided in Appendix H and I.

Equality. `BatchNorm2d` in PyTorch applies batch normalization [25] on a 4D input tensor, with the second dimension as the number of features. We select the parameter `num_features` for the models to predict, with the equality constraint that `num_features` $=$ `input_shapes`$[1]$. Figure 6a shows the results as we increase the difficulty by changing the maximum possible value for each input

dimension. We observe that the DeepSeek Coder models drop from around 0.7∼0.8 to less than 0.5, while GPT-4's performance stays around 0.9 throughout different difficulty levels.

Finding: *Overall, we found that smaller LLMs even struggles with even the simple constraint of copying an existing value, while large state-of-the-art LLMs can maintain its high performance.*

Inequality. `max` in PyTorch computes the maximum value along a dimension. The parameter we target is `dim` with the valid range being `[-rank, rank)`. In Figure 6b, when using greedy decoding, all 4 LLMs achieve close to perfect accuracy. Therefore, we also conduct sampling experiments and present the pass@1 accuracy and diversity in Figure 6b and 6c. For `max` we compute the diversity differently from Section 3.2 (see Appendix G), since the number of possible unique valid outputs is very small. Interestingly, the smaller DeepSeek Coder-1.3b model achieves highest sampling accuracy for `rank=8`, but has the lowest diversity. This is because the smaller model often predicts common values like `1`, whereas the larger model (33b) can explore various correct answers like `-1,2`.

Findings: *We found that larger models are indeed better at capturing the simple inequality constraints and modeling the true probability of various possible values, while smaller models tend to memorize common patterns, leading to less diverse predictions.*

Arithmetic. `reshape` in both PyTorch and NumPy attempts to rearrange the dimensions in the `input_data`, with the constraint being $\prod_i$ `input_shapes`$[i] == \prod_j$ `new_shape`$[j]$. Since we found that it is common for the LLMs to simply predict the same shape or a permutation of the original, we add an additional constraint: we specify the first dimension of the `new_shape` to be different from any dimensions in `input_shapes`. Figure 6d shows the results as we vary the ranks of the `input_data` for PyTorch (similar trend in NumPy). We observe that most LLMs in the beginning perform well; however, as the difficulty increases, their performance drastically lowers. Meanwhile, GPT-4-Turbo performance does not drop even with more difficult inputs. We found the reason is that GPT-4-Turbo tends to always predict the special `-1` value for `reshape` where the `new_shape` will be automatically inferred by the library. Figure 6d showcases this exact phenomenon in PyTorch (similar trend as NumPy) where dotted lines present the accuracy of any outputs without `-1`. We see that now even GPT-4-Turbo struggles in generating valid parameters without using the `-1` crutch for the constraint.

`Conv2d` in PyTorch applies a 2D convolution over a 4D input tensor. The LLMs are asked to predict the parameter `groups`, where they have to divide both `in_channels` and `out_channels` evenly. The default value for `groups` is the trivial 1 (and therefore always valid). To ensure that there is at least one non-trivial value for `groups`, we randomly sample `in_channels` and `out_channels` within the value range such that their greatest common divisor is greater than 1. Figure 6e shows that the accuracy steadily drops as we increase the magnitude of values: even GPT-4-Turbo can only solve ∼24% of the hardest subset of problems, which other models drop below 14% for the same problems.

`Fold` in PyTorch aims to combine an array of sliding local blocks into a large containing tensor. The constraint required for `fold` is the most complex out of all studied APIs where the LLM tries to generate a `k_size` tuple, and the product of the tuple must divide the 2nd index of the `input_shapes` evenly. Furthermore, it also needs to satisfy a complex equation over multiple parameters as shown in Figure 6f. We use the default values for all parameters other than `out_size` and ask LLMs to produce the correct `k_size`. Shown in Figure 6f, due to the complexity of the constraint, even on the lowest difficulty with small values, LLMs achieve relatively poor accuracy compared to other APIs. As we increase the values, the accuracy drops to nearly 0%. This highlights the high degree of difficulty in many DS APIs which current LLMs cannot reliably solve.

Findings: *Arithmetic parameter constraints in DS APIs are extremely challenging for all LLMs. Our results show that current state-of-the-art LLMs cannot effectively solve such complex constraints with their performance drops drastically and even sometimes drops to zero as we increase the difficulty.*

Set-related. `transpose` in NumPy attempts to rearrange/transpose the `input_data` according to the given `new_dim`. In `transpose`, the constraint is that the model-predicted `new_dim` must be a permutation of the original dimensions in `input_data`. We found that the LLMs tend to predict very simple permutations; as such, similar to `reshape`, we directly provide the first dimension of `new_dim` to increase the difficulty. We see that in Figure 6g, LLMs generally perform well on solving this constraint, and their performance improves with larger model sizes. Interestingly, the lowest difficulty of `rank` = 2 has a drop in performance. We theorize that this is because when the `rank` is 2, it is

Table 2: DSEVAL benchmark result. Each column shows both the accuracy/diversity and ranking (⌛).

	Size	PyTorch Easy Acc (⌛)	PyTorch Medium Acc (⌛)	PyTorch Hard Acc (⌛)	PyTorch Div (⌛)	NumPy Easy Acc (⌛)	NumPy Medium Acc (⌛)	NumPy Hard Acc (⌛)	NumPy Div (⌛)
GPT-4-Turbo	NA	77.2 (1)	66.2 (1)	57.5 (1)	- (-)	95.3 (1)	85.1 (1)	71.4 (1)	- (-)
DeepSeek	33b	64.7 (5)	41.5 (4)	28.2 (5)	25.8 (6)	78.5 (3)	57.0 (2)	48.8 (3)	20.9 (1)
	6.7b	66.2 (3)	39.8 (5)	33.4 (4)	38.8 (4)	73.3 (5)	45.8 (8)	35.6 (7)	17.6 (7)
	1.3b	59.0 (8)	34.4 (6)	26.8 (6)	36.2 (5)	63.4 (8)	46.3 (7)	30.5 (8)	17.8 (6)
CodeLlama	13b	64.7 (6)	44.6 (3)	34.8 (3)	39.2 (3)	74.4 (4)	48.5 (6)	36.8 (6)	18.9 (3)
	7b	62.6 (7)	32.7 (8)	13.8 (8)	21.2 (7)	67.1 (7)	53.2 (5)	45.4 (5)	18.7 (4)
StarCoder	15b	65.6 (4)	46.3 (2)	39.2 (2)	39.9 (2)	70.8 (6)	56.7 (3)	51.5 (2)	18.3 (5)
CodeQwen1.5	7b	67.5 (2)	33.2 (7)	25.2 (7)	53.2 (1)	80.0 (2)	54.7 (4)	47.1 (4)	19.3 (2)

more common to directly call `transpose()` without any additional arguments. Therefore, the LLMs struggle a bit when given this unnatural task when asked to predict `new_dim` in low ranks.

Findings: *We found that LLMs generally perform well across the set-related constraints, and their performance scales with increasing model sizes. However, they still struggle with uncommon or unnatural inputs that are no commonly seen during training.*

Instruction-tuned models. We additionally investigate the performance of instruction-tuned (IT) LLMs [59] with chain-of-thought (CoT) prompting [51]. Due to computational limitations, we selected 3 constraints from PyTorch on which GPT-4-Turbo (without CoT) performs poorly for this experiment and analysis. The detailed experimental setup is described in Appendix I. Recall that for `Conv2d`, the task is essentially to predict `groups` such that it is a common divisor of two integers. As we observe that some models tend to predict a trivial answer `1`, we specifically mention "Don't set `groups=1`" in the prompt and consider such answer as invalid in evaluation. From Figure 6h, we observe that GPT-4-Turbo with CoT performs well at this non-trivial task, maintaining over 85% accuracy even with values up to 255. By contrast, the best open-source model can only solve 22%! This shows that although models like CodeQwen achieves close performance to GPT-4-Turbo on existing popular benchmarks like HUMANEVAL [10], there is still a huge gap in terms of coding and math reasoning ability between GPT-4-Turbo and other open-source models. Meanwhile, when we use the same setup on the extremely difficult constraint in `Fold`, we see that even GPT-4-Turbo fails to perform well (less than 5% accuracy in later difficulty settings). This demonstrates that while CoT prompting may elicit better performance in constraints like in `Conv2d`, it still cannot effectively handle other more complex arithmetic constraints. In addition to CoT, we also test ReAct [57], another prompting strategy to elicit more reasoning process from LLMs. We observe that while ReAct can perform better than CoT, it still fails to solve more complex arithmetic constraints (detailed in Appendix J). Additionally, we attempt to include API documentation in prompts, but found that this does not always improve performance on our tasks (detailed in Appendix K).

4.4 DSEVAL: A public benchmark for numerical DS API constraints

Based on the above findings, we further construct a public benchmark – DSEVAL with the same individual parameter prediction setting and the same representative set of APIs as studied in the Section 4.3. For each API in the benchmark, there are 7 different difficulty settings (grouped as 2 *easy*, 3 *medium*, and 2 *hard* ones) and each with 200 randomly created problems. In total, this gives us 19,600 problems in DSEVAL to extensively evaluate the performance of different LLMs.

Table 2 shows the accuracy and diversity of all 8 models. First, we observe that the LLMs' accuracy drops when increasing the difficulty levels on the benchmark problems. This is also reflected by prior results where LLMs across the board struggle with more difficult problems. Next, we see that GPT-4-Turbo consistently achieves the highest accuracy across all difficulty levels, showing the gap between state-of-the-art proprietary models and other open-source LLMs. Furthermore, we observe some interesting ranking changes across difficulty levels. For example, while CodeQwen1.5 [3] achieves the second-best performance in the lowest difficulty level, its performance drops substantially on the medium and hard problems (second worst on PyTorch medium and hard). Other models like StarCoder [31] improve their relative performance and achieve higher ranking on more difficult

problems, showing that different LLMs can perform differently depending on the input and constraint required to satisfy.

We also study the diversity (see Appendix G for more details) of the LLM outputs, except we do not study GPT-4-Turbo due to its cost. Interestingly, LLMs which achieve high ranking in accuracy do not necessarily perform well in generating diverse correct solutions. This indicates that certain LLMs generate similar solutions to satisfy the constraint, without paying attention to the specific context. Therefore, they are not suitable for tasks like fuzz testing [16] which requires efficiently exploring a large solution space, or for tasks involving uncommon API usage. We further categorize some common mistakes made by LLMs on DSEVAL and provide additional insights in Appendix E. Overall, DSEVAL serves as the first benchmark to systematically evaluate the performance of LLMs on satisfying complex numeric API constraints for popular DS libraries and can be extended to support additional APIs and DS libraries.

5 Related work

LLMs for code. LLMs have made remarkable advancements in a wide range of coding tasks, including code synthesis [60, 10, 2], debugging [11, 8], repair [53, 54, 7], and analysis [36, 56, 55]. Notably, recent works [29, 16] also demonstrated LLMs' effectiveness in synthesizing DS code, which requires programming proficiency in DS APIs from specialized libraries such as NumPy [38] and PyTorch [41]. Trained on billions of code including such DS code, LLMs, such as StarCoder [31] and DeepSeek Coder [22], have been extensively evaluated on DS code synthesis tasks. However, no prior study has systematically examined whether LLMs can indeed understand numerical API constraints of these scientific libraries instead of just memorizing the trained data [14].

Coding benchmarks for LLMs. Most code generation benchmarks [10, 33, 2, 22] are formulated with a natural language description and tests to verify the functional correctness of LLM-generated code. However, these benchmarks mostly target general-purpose code. To access LLM code generation for DS tasks, DS-1000 [29] is created by collecting real DS problems from StackOverflow, and ARCADE [58] evaluates LLMs' ability to solve multiple interrelated problems within DS notebooks. Compared to existing DS benchmarks, our study explores different granularity levels to systematically evaluate to what extent LLMs can implicitly learn DS APIs' numeric parameter constraints.

Math reasoning of LLMs. To evaluate LLMs' arithmetic reasoning performance, GSM8K and other benchmarks [12, 42, 35, 24, 28] construct math problems in natural language requiring mathematical computations to solve. Compared to these existing benchmarks, problems designed in our study implicitly encode the arithmetic logic inside the DS library API, and thus can evaluate the LLMs' capability in understanding and solving numerical API constraints in the important DS libraries.

6 Conclusion

In this paper, we present the first systematic study on how LLMs understand the numerical API constraints for important DS libraries. Our study results show that current LLMs often memoize common patterns rather than truly understanding the actual numerical API constraints. Moreover, GPT-4-Turbo largely outperforms other open-source models and can well understand some simple arithmetic constraints using CoT. Based on our finding results, we also constructed DSEVAL, the first benchmark (with 19,000 problems) for systematically evaluating LLMs' capabilities in understanding the important numerical API constraints for popular DS libraries (such as PyTorch and NumPy).

Acknowledgments and Disclosure of Funding

This work was partially supported by NSF grant CCF-2131943 and Kwai Inc. This project is supported, in part, by funding from Two Sigma Investments, LP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Two Sigma Investments, LP.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[4] Silvio Barra, Salvatore M. Carta, Andrea Corriga, Alessandro Sebastian Podda, and Diego Reforgiato Recupero. Deep learning and time series-to-image encoding for financial forecasting. *IEEE/CAA Journal of Automatica Sinica*, 7:683–692, 2020. URL `https://api.semanticscholar.org/CorpusID:218468218`.

[5] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. *Handbook of model checking*, pp. 305–343, 2018.

[6] 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.

[7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[8] Nghi Bui, Yue Wang, and Steven C.H. Hoi. Detect-localize-repair: A unified framework for learning to debug with CodeT5. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 812–823, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.57. URL `https://aclanthology.org/2022.findings-emnlp.57`.

[9] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2722–2730, 2015. doi: 10.1109/ICCV.2015.312.

[10] 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*, 2021.

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023. URL `https://api.semanticscholar.org/CorpusID:258059885`.

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

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[14] Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. Benchmark probing: Investigating data leakage in large language models. In *NeurIPS 2023 Workshop on Backdoors in Deep Learning-The Good, the Bad, and the Ugly*, 2023.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pp. 423–435, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598067. URL https://doi.org/10.1145/3597926.3598067.

[17] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623343. URL https://doi.org/10.1145/3597503.3623343.

[18] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28:653–664, 2017. URL https://api.semanticscholar.org/CorpusID:9398383.

[19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 1469–1481. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00128. URL https://doi.org/10.1109/ICSE48619.2023.00128.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[21] Ken Gu, Madeleine Grunde-McLaughlin, Andrew McNutt, Jeffrey Heer, and Tim Althoff. How do data analysts respond to ai assistance? a wizard-of-oz study. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–22, 2024.

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[23] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: automated testing for deep learning frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pp. 486–498, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416571. URL https://doi.org/10.1145/3324884.3416571.

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

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[26] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 510–520, 2019.

[27] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2022. doi: 10.1109/TITS.2021.3054625.

[28] Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. MAWPS: A math word problem repository. In Kevin Knight, Ani Nenkova, and Owen Rambow (eds.), *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1152–1157, San Diego, California, June 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1136. URL https://aclanthology.org/N16-1136.

[29] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

[30] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen, and Shing-Chi Cheung. Comet: Coverage-guided model generation for deep learning library testing. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023. ISSN 1049-331X. doi: 10.1145/3583566. URL https://doi.org/10.1145/3583566.

[31] 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.

[32] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 530–543, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575707. URL https://doi.org/10.1145/3575693.3575707.

[33] 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. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

[34] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[35] Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *International Conference on Learning Representations (ICLR)*, 2023.

[36] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models, 2023.

[37] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*, 2022. URL https://api.semanticscholar.org/CorpusID:252668917.

[38] Numpy. The fundamental package for scientific computing with python. https://numpy.org, Accessed: May, 2024.

[39] Numpy. Numpy documentation. https://numpy.org/doc/, Accessed: May, 2024.

[40] Numpy. Numpy unit tests. https://github.com/numpy/numpy/tree/main/numpy/tests, Accessed: May, 2024.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[42] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are nlp models really able to solve simple math word problems?, 2021.

[43] PyTorch. Pytorch documentation. https://pytorch.org/docs/stable/index.html, Accessed: May, 2024.

[44] PyTorch. Pytorch unit tests. https://github.com/pytorch/pytorch/tree/main/test, Accessed: May, 2024.

[45] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, IUI '23, pp. 491–514, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701061. doi: 10.1145/3581641.3584037. URL https://doi.org/10.1145/3581641.3584037.

[46] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving, 2016.

[47] Yiyin Shen, Xinyi Ai, Adalbert Gerald Soosai Raj, Rogers Jeffrey Leo John, and Meenakshi Syamkumar. Implications of chatgpt for data science education. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pp. 1230–1236, 2024.

[48] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Using large language models to generate junit tests: An empirical study, 2024.

[49] Wil Van Der Aalst and Wil van der Aalst. *Data science in action*. Springer, 2016.

[50] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 995–1007, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510041. URL https://doi.org/10.1145/3510003.3510041.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[52] Wikipedia contributors. Plagiarism — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/wiki/Hellinger_distance. [Online; accessed 20-May-2024].

[53] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, 2022.

[54] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 1482–1494. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00129. URL https://doi.org/10.1109/ICSE48619.2023.00129.

[55] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*, 2023.

[56] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735, 2024.

[57] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

[58] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 126–173, 2023.

[59] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792*, 2023.

[60] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Lr8cOOtYbfL.