arXiv:2402.08073v1 [cs.LG] 12 Feb 2024

Grounding Data Science Code Generation with Input-Output Specifications

Yeming Wen [1] **Pengcheng Yin** [2] **Kensen Shi** [2] **Henryk Michalewski** [2] **Swarat Chaudhuri** [1] **Alex Polozov** [2]

Abstract

Large language models (LLMs) have recently demonstrated a remarkable ability to generate code from natural language (NL) prompts. However, in the real world, NL is often too ambiguous to capture the true intent behind programming problems, requiring additional input-output (I/O) specifications. Unfortunately, LLMs can have difficulty aligning their outputs with both the NL prompt and the I/O specification. In this paper, we give a way to mitigate this issue in the context of data science programming, where tasks require explicit I/O specifications for clarity. Specifically, we propose GIFT4CODE, a novel approach for the instruction fine-tuning of LLMs with respect to I/O specifications. Our method leverages synthetic data produced by the LLM itself and utilizes execution-derived feedback as a key learning signal. This feedback, in the form of program I/O specifications, is provided to the LLM to facilitate instruction fine-tuning. We evaluated our approach on two challenging data science benchmarks, ARCADE and DS-1000. The results demonstrate a significant improvement in the LLM's ability to generate code that is not only executable but also accurately aligned with user specifications, substantially improving the quality of code generation for complex data science tasks.

1 Introduction

Large language models (LLMs) have recently shown tremendous promise at generating code from natural language prompts (Chen et al., 2021a; Austin et al., 2021; Li et al., 2023; 2022; Nijkamp et al., 2022; Fried et al., 2022; Li et al., 2023). In particular, LLMs trained on code have been shown to excel at solving interview-style coding problems, represented by benchmarks such as HumanEval (Chen et al., 2021a). In these tasks, natural language (NL) "intents" (*e.g.* $\boldsymbol{x}$ in Fig. 1) can succinctly define the programming problem at hand. The objective of the LLM is to produce output that is aligned with this intent.

However, in the real world, not all programming tasks can be easily described using compact NL intents. Among these, data science programming stands out as a vital domain, where data scientists spend considerable time on data wrangling and exploratory data analysis (EDA). Whereas problems in benchmarks like HumanEval involve basic Python types that can be easily tested, data science tasks often involve complex operations on data structures like `Pandas` DataFrames or `PyTorch` Tensors. The inherent ambiguity in expressing these tasks through NL intents (*e.g.* green text in Fig. 1) often leads to misalignment between developers' requirements and the code generated (Yin et al., 2022).

Input-output (I/O) specifications (*e.g.* red text in Fig. 1), ranging from concrete I/O examples to high-level NL summaries, are a natural way to reduce the ambiguity of NL intents (Gulwani et al., 2015; Balog et al., 2016; Jain et al., 2022). Prior to the emergence of LLMs, specifications served as essential problem descriptions in program synthesis (Gulwani, 2016; Devlin et al., 2017; Shi et al., 2020). Real-world synthesis systems like FlashFill are testimony to the adoption and effectiveness of I/O specifications (Gulwani, 2011; Gulwani et al., 2012). In this work, we consider the problem of LLM-based code generation in data science when the prompt provided to the LLM consists of both a NL intent and an additional I/O specification.

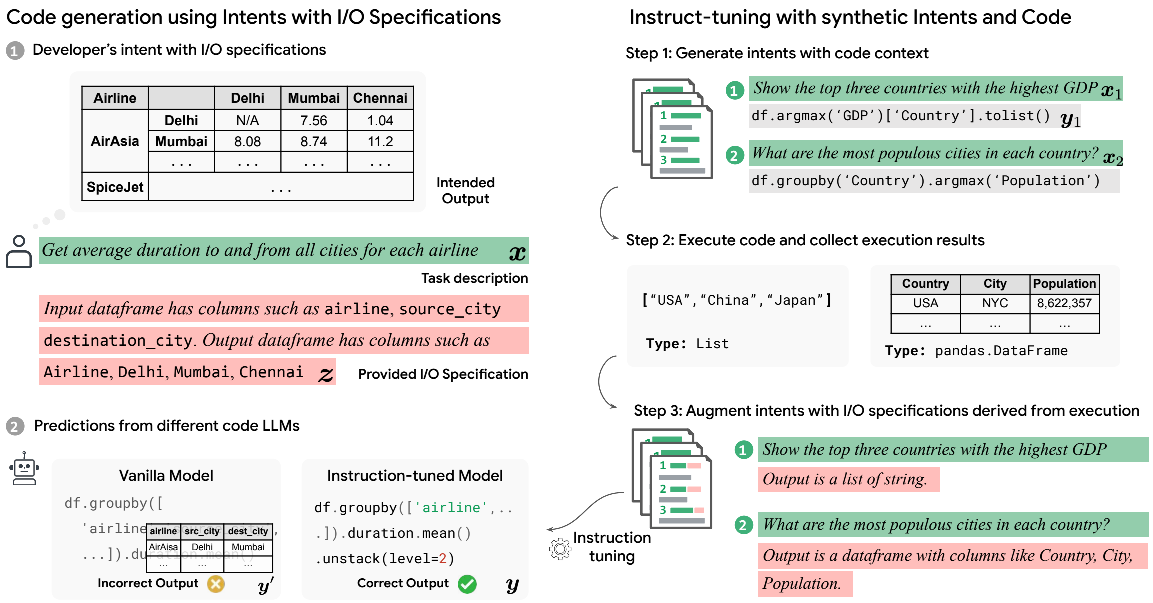
Unfortunately, code LLMs often fail to follow intents with additional semantic constraints like I/O specifications out-of-the-box, leading to plausible solutions that fail to satisfy the provided specifications (*e.g.* $\boldsymbol{y}'$, Fig. 1). This can pose unnecessary burden on developers who are then required to fix the generated code (Bird et al., 2023). Such a misalignment between the user's intent and the model's predictions primarily stems from the *lack* of training data formatted with such specifications.

Instruction fine-tuning has emerged as an effective strategy to tackle the issue of misalignment (Wei et al., 2021; Sanh et al., 2021; Chung et al., 2022). Classical approaches for instruction tuning typically require a substantial amount of parallel labeled data of NL intents and gold model responses. The process of gathering such data is labor-

---

[1]Department of Computer Science, University of Texas at Austin [2]Google Inc.. Correspondence to: Yeming Wen, Work done during Internship at Google. <ywen@utexas.edu>.

Preprint.


Code generation using Intents with I/O Specifications
① Developer's intent with I/O specifications
Airline
Delhi
Mumbai
Chennai
AirAsia
N/A
7.56
1.04
8.08
8.74
11.2
SpiceJet
Intended Output
Get average duration to and from all cities for each airline $\boldsymbol{x}$
Task description
Input dataframe has columns such as airline, source_city destination_city. Output dataframe has columns such as Airline, Delhi, Mumbai, Chennai $\boldsymbol{z}$
Provided I/O Specification
② Predictions from different code LLMs
Vanilla Model
airline
src_city
dest_city
AirAisa
Delhi
Mumbai
Incorrect Output
$\boldsymbol{y}'$
Instruction-tuned Model
Correct Output
$\boldsymbol{y}$
Instruct-tuning with synthetic Intents and Code
Step 1: Generate intents with code context
① Show the top three countries with the highest GDP $\boldsymbol{x}_1$
$\boldsymbol{y}_1$
② What are the most populous cities in each country? $\boldsymbol{x}_2$
Step 2: Execute code and collect execution results
["USA", "China", "Japan"]
Type: List
Country
City
Population
USA
NYC
8,622,357
Type: pandas.DataFrame
Step 3: Augment intents with I/O specifications derived from execution
① Show the top three countries with the highest GDP
Output is a list of string.
② What are the most populous cities in each country?
Output is a dataframe with columns like Country, City, Population.
Instruction tuning

```
df.groupby([
  'airline'...
  ...]).du
```

```
df.groupby(['airline',..
.]).duration.mean()
.unstack(level=2)
```

```
df.argmax('GDP')['Country'].tolist()
```

```
df.groupby('Country').argmax('Population')
```

Figure 1: ***Left*:** Illustration of how users prompt code LLMs with NL intents and I/O specifications to generate code with complex outputs (`pandas Dataframes`). Vanilla code LLMs fail to understand I/O specifications. ***Right*:** Our proposed instruction tuning approach uses synthetic intents and code solutions, where intents are augmented with I/O specifications derived from program execution. Models trained on such synthetic data could better follow a user's intent.

intensive and time-consuming. Recent studies have suggested that generating synthetic instruction-following data using the LLM itself is a promising approach to improve alignment, with empirical success on natural language text generation tasks (Wang et al., 2022a; Honovich et al., 2022a; Taori et al., 2023; Peng et al., 2023, *inter alia*).

In this paper we build upon the recent success of instruction tuning using synthetic data and fine-tune code LLMs to follow NL intents with additional I/O specifications. Unlike existing approaches, our key insight is to leverage program execution for synthetic data generation. First, in contrast to other open-ended text generation tasks where assessing the quality of target responses is challenging, the quality of synthetic code generation data can be easily improved using heuristics such as code executability (Yin et al., 2022). Moreover, from the program execution states one could derive precise and *aligned* I/O specifications that can be included in the intents to supervise a model to follow those extra semantic constraints (Fig. 1, *Right*). In other words, when fine-tuned on such synthetic data, a model learns to *ground* NL task descriptions to program execution states expressed as I/O specifications (Berant et al., 2013).

We apply our grounded instruction fine-tuning for code (GIFT4CODE) method to two challenging natural language to code generation applications: synthesizing complex `pandas` programs in computational notebooks (ARCADE, Yin et al. (2022)) and answering data science questions on Stack Overflow (DS-1000, Lai et al. (2022)). First, we demonstrate the value of leveraging program execution information by showing that strong code LLMs can still be improved after fine-tuning on our synthetic data *without* including any I/O specifications. Then, to further align model predictions to various types of user-provided I/O specifications, we derive those specifications at different levels of abstraction from code execution results. This ranges from concrete input/output examples to NL summaries of target variables (specifications in Fig. 1). By fine-tuning on parallel data of intents with I/O specifications and their target code solutions, the model is better at producing code that is more likely to execute to the desired outcome. Although our experiments focused on the data science domain, it's important to note that the methodology underlying GIFT4CODE is general and adaptable. It can be applied to different domains that require specifications for a precise task description.

2 Problem Formulation

Natural Language to Code Generation Code generation considers the task of translating a developer's natural language intent $\boldsymbol{x}$ into a machine-executable program $\boldsymbol{y}$ (*e.g.* Fig. 1, *Left*). An intent usually contains a succinct and potentially ambiguous task description. For tasks with complex outputs, the intent may also include additional I/O specifications as extra clarifications.[1] Code generation tasks are often contextualized, meaning that an

---

[1] For simplicity, we consider I/O specifications as part of the intent hereafter.

intent is associated with certain *programmatic contexts* $\boldsymbol{c}$ (Iyer et al., 2018), such as the code that a developer has already written in an IDE prior to the intent (*e.g.* `df = pd.read_csv("flights.csv")` for $\boldsymbol{x}$, not shown in Fig. 1). Intuitively, a model needs to leverage both the intent and the programmatic context (*e.g.* variable `df`) to generate a suitable code solution.

Supervised Instruction Tuning for Code LLMs Supervised instruction tuning aims to improve code LLMs by fine-tuning them on parallel data of intents and target code solutions. In this paper we consider automatically synthesizing such parallel data by prompting LLMs using few-shot demonstrations (other approaches are discussed in §5). Specifically, the synthetic dataset consists of examples $\{\langle \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y} \rangle\}$ of intents $\boldsymbol{x}$ with programmatic contexts $\boldsymbol{c}$ and their generated code solutions $\boldsymbol{y}$. The pivotal question then becomes how to create a high-quality synthetic $\{\langle \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y} \rangle\}$ that facilitates the instruction fine-tuning process.

3 GIFT4CODE: Learning to Follow Intents with I/O Specifications

In this section we elaborate on GIFT4CODE, our proposed approach to fine-tune code LLMs to better follow developers' natural language intents along with I/O specifications, using synthetic parallel data. Fig. 1(*Right*) illustrates an overview of GIFT4CODE. We first synthesize a collection of intents with code solutions via few-shot prompting (§3.1), and then execute model-predicted code to derive I/O specifications from execution results (§3.2). Finally, we fine-tune the code LLM to predict code solutions given intents inlined with I/O specifications (§3.3).

3.1 Generating Synthetic Intents and Code Solutions

Programmatic Contexts We initialize a program state given some programmatic context and generate a series of contextualized NL-to-code problems for that context. As an example, the synthetic problems in Fig. 1 (*Right*) could have the contextual code `df = pd.read_csv("world_statistics.csv")`, which initializes the DataFrame variable `df`, subsequently used in the generated synthetic examples. The fact that our problems are contextualized sets our approach apart from existing instruction-tuning methods for text generation models (Wang et al., 2022a; Honovich et al., 2022a), where synthetic examples do not depend on any particular contexts. In our case, we mine those programmatic contexts from real-world code repositories, such as tabular datasets (e.g., `.csv`) used in data science notebooks on Github (§4).

Creating Initial NL Intents Given a programmatic context $\boldsymbol{c}$, we few-shot prompt an LLM to create a sequence of natural language intents $\{\boldsymbol{x}_i\}$ (*e.g.* $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ in Fig. 1(*Right*)). A problem $\boldsymbol{x}_i$ that appears later in the sequence might depend on the earlier ones $\{\boldsymbol{x}_{<i}\}$ (Nijkamp et al., 2022; Yin et al., 2022). To generate NL intents, we use a "generalist" PALM2 LLM (Google, 2023), instead of the code LLM that we aim to improve, since predicting intents conditioned on some context is similar to other text generation tasks, which could be better handled by a LM trained on general-purpose text data (Zelikman et al., 2022). The PALM2 LLM achieves competitive results with GPT-4 on a variety of NL reasoning tasks. Empirically, we observe that the problems generated by this LLM cover a wide range of tasks relevant to the given programmatic context. Readers can refer to Appendix B for examples. Note that those model-predicted intents do not come with I/O specifications yet.

Predicting Code Solutions After generating an intent $\boldsymbol{x}$, we then prompt the code LLM to get a code solution $\boldsymbol{y}$ for $\boldsymbol{x}$ (*e.g.* $\boldsymbol{y}_1$ in Fig. 1(*Right*)). Specifically, a prompt to the LLM is the concatenation of the programmatic context $\boldsymbol{c}$ and the intent $\boldsymbol{x}$, with additional few-shot demonstrations of $\{\langle \boldsymbol{c}', \boldsymbol{x}', \boldsymbol{y}' \rangle\}$. Since many NL intents can be ambiguous and there could exist multiple alternative solutions (*e.g.* without additional I/O specifications, the intent in green in Fig. 1(*Left*) could be answered using tables with different layouts; see more in (Yin et al., 2022)), we therefore draw multiple candidate code solutions $\{\boldsymbol{y}\}$ for each intent. Intuitively, $\{\boldsymbol{y}\}$ could have a variety of alternative solutions for $\boldsymbol{x}$, each leading to different execution results. This equips the model with the capacity to predict code for the same task but with different user-provided I/O specifications.

Improving Quality of Synthetic Data The quality of synthetic data is a fundamental issue of data augmentation in instruction tuning (Wang et al., 2022a; Honovich et al., 2022a), and existing approaches in text generation typically resort to simple and noisy heuristics (*e.g.* rejecting examples with different inputs but the same output). As motivated in §1, for NL-to-code generation, we can reliably enhance the quality of candidate code solutions by leveraging inherent program properties, such as filtering out any code that is not executable given the provided programmatic context.

3.2 Code Execution & Inference of I/O Specifications

Given the set of synthetic problems $\{\langle \boldsymbol{x}, \{\boldsymbol{y}\} \rangle\}$ generated by few-shot prompting, we execute the code for each problem (step 2, Fig. 1(*Right*)) and derive I/O specifications from the execution results as additional semantic constraints to be included in the intents (step 3, Fig. 1(*Right*)).

Specifically, for each candidate solution $\boldsymbol{y}$ of an intent, we first execute its original programmatic context $\boldsymbol{c}$, followed by executing $\boldsymbol{y}$. We trace the execution to collect the set of input and output variables in $\boldsymbol{y}$, denoted as $\{v\}$, which are used to derive I/O specifications (details below). Executing code with arbitrary programmatic contexts collected from

Spec. Type	Description	Example I/O Specification
TypeDesc	Variable type name	Generate a variable with name `df` and type `pandas.DataFrame`
I/O Examples	Concrete I/O examples	Output variable `df`: (see below)
I/O Summary	LLM-generated NL summaries of I/O examples	Given the user intent and the code, the salient columns (at most given 3) in the input dataframe are airline, source_city, destination_city. The output dataframe has columns (at most given 3) such as Delhi, Mumbai, Chennai.

I/O Examples:

```
|Bangalore(float)|Chennai(float)|Delhi(float)|Hyderabad
(float)|Kolkata(float)|Hyderabad(float)|Kolkata(float)|...
|-----|-----|-----|-----|-----|-----|
| nan | 1.04 | 8.08 | 3.62 | 7.56 | 7.56 | 8.32 |
| 1.18 | nan | 11.96 | 6.80 | 6.31 | 8.75 |
| 8.46 | 11.10 | nan | 9.19 | 9.52 | 10.32 |
```

Table 1: Types of I/O specifications proposed in this work at different levels of abstraction. Example specifications are for the intent in Fig. 1(*Left*). Only the output specifications for I/O Examples are shown for brevity.

the wild is highly non-trivial due to issues such as library dependency. However, the use of synthetic data alleviates the need for a complex environment setup.

Given the set of input and output variables extracted from execution results, we formulate an I/O specification, denoted as $\boldsymbol{z}$, which serves as additional information to augment a developer's intent, thereby providing a more comprehensive problem description. The level of detail and the style of these I/O specifications can vary based on the complexity of the problem and the developer's preferences. In this work, we investigate three distinct types of I/O specifications, each characterized by its own linguistic style and level of abstraction, as illustrated in Tab. 1.

First, as a simple baseline, we utilize the variable type (TypeDesc, Tab. 1) as the I/O specification. Next, we incorporate the concrete values of the input/output variables into the specification, which we refer to as I/O Examples. This is reminiscent of classical program synthesis using I/O examples (Gulwani et al., 2012; Alur et al., 2013; Balog et al., 2016). However, in our scenario, these I/O examples are used in conjunction with natural language (NL) intents to define the problem, in line with (Jain et al., 2022). Given that the majority of the problems in our synthetic dataset involve complex Python objects like `pandas DataFrames`, we simplify the I/O specification to include only partial variable states (*e.g.* by excluding some rows and columns in large `DataFrames`). Please refer to Appendix C for details.

In our effort to generate a more natural variety of I/O specifications that closely resemble the style of specifications in developers' NL intents, we employ an LLM to summarize the values of input/output variables $\{v\}$ into a succinct natural language description $\boldsymbol{z}$ (I/O Summary). Intuitively, the NL I/O summary includes salient information in the variables that can best clarify the original intent (*e.g.* the subset of columns in a `DataFrame` that are most relevant to solve a problem, as in Tab. 1, *Bottom*).

Specifically, we few-shot prompt the "generalist" PaLM2 to generate $\boldsymbol{z}$, using information from its programmatic context $\boldsymbol{c}$, the intent $\boldsymbol{x}$, the code solution $\boldsymbol{y}$, as well as I/O variables $\{v\}$, *i.e.* $\boldsymbol{z} \sim P_{\text{LLM}}(\cdot \mid \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y}, \{v\})$. We then update the intent $\boldsymbol{x}$ by appending $\boldsymbol{z}$ to it. The few-shot exemplars used for prompting cover example I/O summaries for various types of Python objects, such as nested container types (*e.g.* nested `dicts`), along with more complex objects like `Pandas DataFrames` and `PyTorch` or `Tensorflow Tensors`. See Appendix C for additional details.

3.3 Fine-tuning Code LLMs to Follow Intents with I/O Specifications

Our approach, GIFT4CODE, aims to fine-tune code LLMs to generate code that adheres closely to the desired intents which are supplemented by I/O specifications. In our synthetic training data, each example $\langle \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y} \rangle$ consists of a programmatic context $\boldsymbol{c}$, an intent $\boldsymbol{x}$ augmented with I/O specifications, and the corresponding code solution $\boldsymbol{y}$. During fine-tuning, the code LLM learns to generate code that not only satisfies the provided intents but also respects the specified I/O constraints, while leveraging any relevant information in the programmatic contexts. In other words, we optimize $P_{\text{LLM}}(\boldsymbol{y} \mid \boldsymbol{c}, \boldsymbol{x})$. It is worth noting that the code LLM that undergoes this optimization is different from the "generalist" PaLM2 LLM employed to generate the NL intents and I/O specification $\boldsymbol{z}$. [2]

4 Experiments

The core research question explored in this section is whether GIFT4CODE enhances the LLM's ability to follow developers' NL intents with ***complex*** I/O specifications. While common code generation benchmarks like HumanEval (Chen et al., 2021a) and MBPP (Austin et al.,

---

[2] The "generalist" LLM, such as PaLM2 or GPT-4, is typically accessible only through an API.

2021) feature simple algorithmic tasks (e.g., sorting) utilizing basic Python data types (e.g., lists), thus allowing for the use of concrete I/O examples as specifications, they lack the diverse and complex I/O specifications that we aim to explore. For more open-ended tasks in the data science domain, the output data type is more complex and diverse (e.g., `pandas DataFrames`, `pytorch Tensors`). Hence, we applied our method to two different data science code generation applications.

ARCADE (Yin et al., 2022) is a benchmark of natural language to code generation in interactive data science notebooks. Each evaluation notebook consists of a series of interrelated NL-to-code problems in data wrangling (*e.g. "Minmax normalize numeric columns"*) and exploratory data analysis (*e.g.* intents in Fig. 1) using the `pandas` library. ARCADE features succinct NL intents to reflect the style of ephemeral queries from developers when prompting LLMs for code completion. More than 50% of the dataset's problems are under-specified, which means that additional I/O specifications could provide extra clarification. To construct programmatic contexts for synthetic training data generation, we scraped 7,500 CSV files that are used in public Jupyter notebooks. Each context contains a DataFrame import statement, for example, `df = pd.read_csv(·)`, followed by an NL description of the DataFrame to help the LLM understand its content. We generated 6 intents for each programmatic context and sampled 5 candidate code solutions for each intent. Roughly 60% of the code samples were executable. After filtering based on executability and API diversity (§3.1), we obtained around $20K$ synthetic examples for instruction fine-tuning.

Our synthetic data only comprises pairs of questions and code samples which lack rich context. To avoid regression in context understanding during instruction fine-tuning, we crafted a mixture dataset which combines the synthetic data and the Python data used to fine-tune the code LLM. Note that this Python data does not contain any execution-based signals or I/O specifications. After approximately 1,500 instruction tuning steps with a batch size of 64, the model reaches its optimal performance. This process consumed about 1.5 epochs of our synthetic dataset.

DS-1000 (Lai et al., 2022) is a benchmark of data science problems sourced from Stack Overflow (SO). Compared to ARCADE, problems in DS-1000 feature a wider variety of I/O types, such as `numpy/scipy Arrays` and `pytorch/tensorflow Tensors`, making it particularly appealing to evaluate our instruction tuning approach aimed at generating code following I/O specifications. However, in contrast to ARCADE which features succinct NL intents, DS-1000 follows the typical style of detailed problem descriptions found in SO posts. These elaborate descriptions often include additional information such as task background and descriptions of unsuccessful attempts with an average length of 140 words. Given that such elaborate intents may not reflect the style of developers' prompts to code LLMs, we do not focus on generating intents with similar styles. Instead, we held-out 500 problems in DS-1000 and use their annotated intents as training data, while evaluating on the remaining problems.[3]

4.1 Setup

Base Code LLM We use a PALM based, decoder-only code language model with 62B parameters (Chowdhery et al., 2022). The model was first pre-trained on a collection of 1.3T tokens of web documents and github code data, and was then fine-tuned on a disjoint set of 64B Python code tokens together with 10B tokens from Python Jupyter notebooks. The fine-tuning process, including the GIFT4CODE procedure, can be conducted through the fine-tuning API on Google cloud Vertex AI. Notably, this base model has strong performance in data science programming tasks, outperforming the CodeLLama 34B Python model (Rozière et al., 2023) on ARCADE, as evidenced in Tab. 2 [4].

Learning Methods We evaluated the performance of both the baseline and instruction-tuned models across a range of data formats, as shown in Tab. 2. For each I/O specification type, we augmented the intents and few-shot exemplars with specifications of the corresponding type. Similarly, at test time, we augmented the intents with the same type of I/O specifications. For the few-shot prompting, we manually created exemplars for all types of specifications. These exemplars were prepended to the prompt when querying the LLM for code generation.

Simulate Noisy I/O Specifications at Test Time At testing time, the generation of I/O Summary underwent a minor modification from the process detailed in §3.2. We remove the concrete input/output variable states $\{v\}$ to produce noisy I/O summaries, simulating scenarios where users might give noisy I/O specifications (Devlin et al., 2017). We illustrate an example in Appendix C where the LLM generates an imperfect specification. While the "generalist" LLM (PALM2) uses the code solution to generate noisy I/O summaries, we remark that the code LLM, which we aim to evaluate, does not have access to the ground truth solution. In other words, the "generalist" LLM acts merely as a "data labeler" to create I/O summaries in prompts in order to construct the evaluation dataset. It is also a common practice in program synthesis to derive specifications from

---

[3] We only use the annotated intents, while the code solutions and I/O specifications are still predicted by the LLM. We ensure the training and evaluation problems are disjoint and from different Stack Overflow posts.

[4] StarCoder 15B and CodeLLama 34B have slightly higher *pass*@1 on DS-1000 compared to numbers in Ni et al. (2023b).

Methods	**Spec. Type**	**Arcade**				**DS-1000**
		pass@5		*pass*@20		*pass*@1
		No Context	Full Context	No Context	Full Context	
Zero-shot Prompting						
StarCoder 15B	no spec.	11.75	22.38	17.24	32.52	21.52
CodeLlama 34B	no spec.	12.27	23.74	19.10	36.20	22.56
Code LLM 62B	no spec.	12.45	27.75	19.85	37.47	22.62
+ Gift4Code	no spec.	20.78	34.33	33.40	46.94	24.56
Few-shot Prompting						
Code LLM 62B	TypeDesc	16.58	29.68	29.68	42.30	25.90
+ Gift4Code	TypeDesc	21.52	36.73	33.58	48.61	27.35
Code LLM 62B	I/O Examples	19.85	33.77	30.79	43.23	26.41
+ Gift4Code	I/O Examples	25.23	42.30	38.03	53.99	28.66
Code LLM 62B	I/O Summary	23.75	37.11	34.50	46.75	26.25
+ Gift4Code	I/O Summary	**28.01**	**43.79**	**43.04**	**55.47**	**29.34**

Table 2: $pass@k$ on Arcade and DS-1000. For each type of I/O specification in Tab. 1 (*e.g.* +I/O Summary), intents are augmented with I/O specifications of that type (*e.g.* intents inline with I/O summary) in fine-tuning data or few-shot exemplars. At test time, input intents use the same type of I/O specifications.

ground truth solutions, which then serve as the sole input to the model during its evaluation (Balog et al., 2016).

Metrics Developing unit tests for data science programming tasks is often non-trivial. For example, creating a test for a data wrangling task like "What are the most populous cities in each country" (as shown in Fig. 1) is not feasible. In the Arcade dataset, each question is accompanied by an annotated program. Then the evaluation uses a set of heuristics to approximate whether the execution output of a predicted program and the annotated reference are functionally equivalent, by comparing their outputs. On the DS-1000 dataset, evaluation relies on the provided tests.

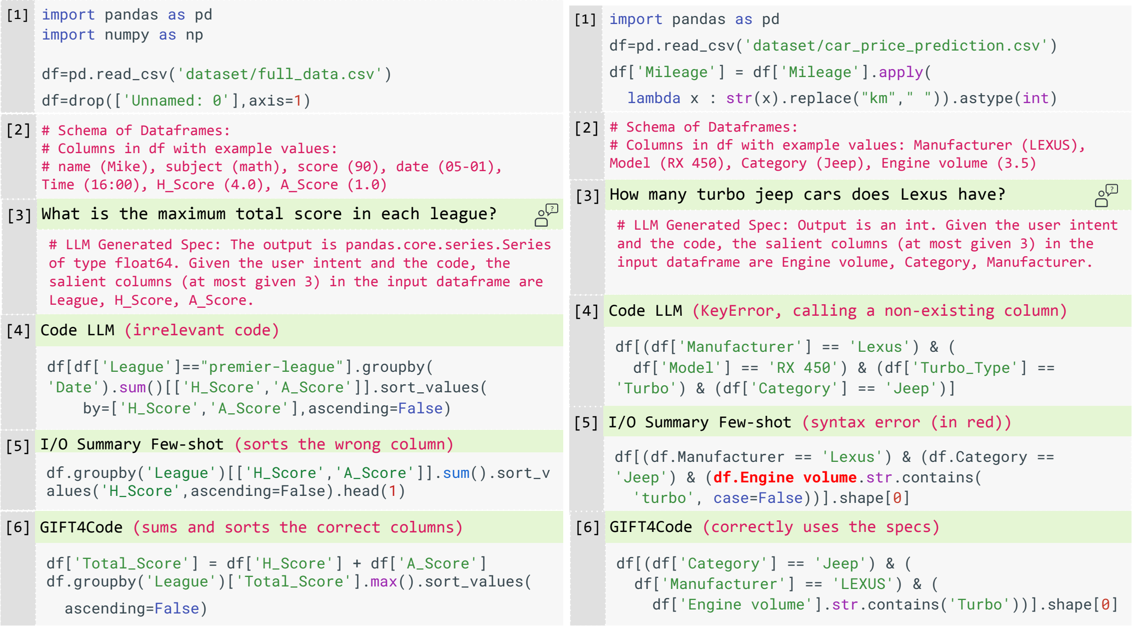
We adopted the $pass@k$ metrics as defined in Chen et al. (2021a); Austin et al. (2021), which is calculated as the fraction of problems with at least one correct sample given k samples. Following Yin et al. (2022), we drew 50 samples to calculate $pass@5$ and $pass@20$ to reduce the variance in Arcade. Similar to Lai et al. (2022), we drew 40 samples to calculate $pass@1$ on DS-1000. Consistent with the original works' settings, the sampling temperature was set to 0.8 for Arcade and to 0.2 for DS-1000 respectively.

4.2 Main Results

Tab. 2 presents the $pass@k$ results on Arcade and DS-1000. We evaluate both few-shot prompting and fine-tuning with synthetic data. Specifically, for Arcade we evaluate on two versions of the dataset. First, we consider the original version where an intent is prefixed by prior notebook cells as its programmatic context (**Full Context**), as well as a **No Context** ablation to simulate the scenario where users query a code LLM using an intent without any context. This no-context setting is more challenging, where the zero-shot performance of the base code LLM is nearly halved. The standard errors in all cells of the table are less than 0.5%, and are excluded for clarity in presentation.

In our few-shot prompting experiments, we observe that $pass@k$ generally improves with more detailed I/O specifications. Interestingly, on Arcade, the improvements from prompting using I/O specifications compared to the baseline where no I/O specifications were used (**no spec**), are more notable in the more challenging no-context scenario (*e.g.* $12.45 \mapsto 23.75$ *v.s.* $27.75 \mapsto 37.11$ for +I/O Summary). This trend suggests that additional specifications could provide more valuable clarifications when adequate programmatic contexts are lacking.

Next, we fine-tune the base code LLM using our synthetic parallel data with different types of I/O specifications. Even without using any I/O specifications in the synthetic data, Gift4Code already registers significant improvements on Arcade ($pass@20$: $37.47 \mapsto 46.94$). The model-predicted code solutions are filtered using executability heuristics, which helps improve the quality of the synthetic data, and a model fine-tuned on such data could generally be better at following users' intents, even without I/O specifications. Moreover, by fine-tuning the model to follow intents with additional I/O specifications, we observe significantly better results. With I/O summaries (+I/O Summary), Gift4Code improves $pass@20$ from 46.75 to 55.47. We also remark that instruction fine-tuning using natural language I/O summaries yields the best results on both datasets. Intuitively, those I/O summaries could encode salient information in target input and output variables through natural language descriptions, which could make it easier for the model to capture patterns in the data.



```
[1] import pandas as pd
    import numpy as np

    df=pd.read_csv('dataset/full_data.csv')
    df=drop(['Unnamed: 0'],axis=1)
[2] # Schema of Dataframes:
    # Columns in df with example values:
    # name (Mike), subject (math), score (90), date (05-01),
    Time (16:00), H_Score (4.0), A_Score (1.0)
[3] What is the maximum total score in each league?
     # LLM Generated Spec: The output is pandas.core.series.Series
     of type float64. Given the user intent and the code, the
     salient columns (at most given 3) in the input dataframe are
     League, H_Score, A_Score.
[4] Code LLM (irrelevant code)
     df[df['League']=="premier-league"].groupby(
     'Date').sum()[['H_Score','A_Score']].sort_values(
         by=['H_Score','A_Score'],ascending=False)
[5] I/O Summary Few-shot (sorts the wrong column)
     df.groupby('League')[['H_Score','A_Score']].sum().sort_v
     alues('H_Score',ascending=False).head(1)
[6] GIFT4Code (sums and sorts the correct columns)
     df['Total_Score'] = df['H_Score'] + df['A_Score']
     df.groupby('League')['Total_Score'].max().sort_values(
       ascending=False)
```

```
[1] import pandas as pd
    df=pd.read_csv('dataset/car_price_prediction.csv')
    df['Mileage'] = df['Mileage'].apply(
      lambda x : str(x).replace("km"," ")).astype(int)
[2] # Schema of Dataframes:
    # Columns in df with example values: Manufacturer (LEXUS),
    Model (RX 450), Category (Jeep), Engine volume (3.5)
[3] How many turbo jeep cars does Lexus have?
     # LLM Generated Spec: Output is an int. Given the user intent
     and the code, the salient columns (at most given 3) in the
     input dataframe are Engine volume, Category, Manufacturer.
[4] Code LLM (KeyError, calling a non-existing column)
     df[(df['Manufacturer'] == 'Lexus') & (
       df['Model'] == 'RX 450') & (df['Turbo_Type'] ==
     'Turbo') & (df['Category'] == 'Jeep')]
[5] I/O Summary Few-shot (syntax error (in red))
     df[(df.Manufacturer == 'Lexus') & (df.Category ==
     'Jeep') & (df.Engine volume.str.contains(
       'turbo', case=False))].shape[0]
[6] GIFT4Code (correctly uses the specs)
     df[(df['Category'] == 'Jeep') & (
       df['Manufacturer'] == 'LEXUS') & (
         df['Engine volume'].str.contains('Turbo'))].shape[0]
```

Figure 2: Two examples on ARCADE. **Left**: the base code LLM does not follow the intent to group the “League” column. Its few-shot prompting variant failed to correctly utilize the specifications. GIFT4CODE’s solution aligns to the user’s intent. **Right**: code LLM tries to call a non-exsiting column, leading to a `KeyError`. Its few-shot variant follows the specifications incorrectly, leading to the syntax error. GIFT4CODE generates correct solution. The prompt always includes dataframe schema, regardless of whether specifications are provided.

4.3 Qualitative Analysis

To gain deeper insights into the behavior of the different models, we present a qualitative analysis of the baseline model, its few-shot prompting variant with LLM specifications (I/O Summary), and our proposed GIFT4CODE approach. We illustrate this analysis using two examples from the ARCADE dataset, as shown in Fig. 2.

In the first example (Fig. 2, *Left*), the base code LLM (cell 4) fails to group the “League” column as requested in the user’s intent. Note that the provided solution is still executable so it cannot be filtered by executability heuristics. The few-shot prompting variant with I/O summary (cell 5) also fails here. It struggles to correctly utilize these specifications despite selecting the correct salient columns, leading to an output that does not meet the user’s requirement either. In contrast, the output from GIFT4CODE (cell 6) successfully generates a solution which computes the sum of the two salient columns then sorts the result, effectively utilizing the provided specifications and adhering to the user’s intent.

The second example (Fig. 2, *Right*) further underscores the advantages of GIFT4CODE. The baseline code LLM (cell 4) attempts to call a non-existing column (`Turbo_Type`), leading to a `KeyError`. This represents a failure mode that the model tries to adhere a user’s intent but generates an inexecutable solution that refers to incorrect input variables due to lack of I/O specifications. The few-shot prompting variant (cell 5) presents another interesting failure mode. While the model is trying to follow the additional I/O specification (presumably because of the few-shot demonstrations) by referring to the `Engine volume` column in the specification, it fails to generate a syntactically correct code snippet (`df.Engine volume`). It is important to note that this is a common failure mode of the few-shot prompting model, as we explain in Fig. 3 later. Once again, GIFT4CODE outperforms other settings, generating solutions that answer the natural language question while following specifications.

4.4 Execution Rate vs *Pass*@k

We delve further into the experimental results to examine the relationship between executability and the quality of the generated code. Surprisingly, we observe that a model with a higher execution rate does not necessarily produce better code. Fig. 3 plots the frequency of common error types alongside the code execution rates of different models’ predictions. The baseline code LLM, despite generating a substantial amount of executable code (higher ★), often produces incorrect (irrelevant) solutions, leading to a high executable rate but low *pass*@k accuracies (Tab. 2). This suggests that a model’s ability to generate executable code does not necessarily indicate its competence in generating semantically correct code that aligns with the user’s intent. This insight is further evidenced when we fine-tune the model on synthetic data without I/O specifications, labeled

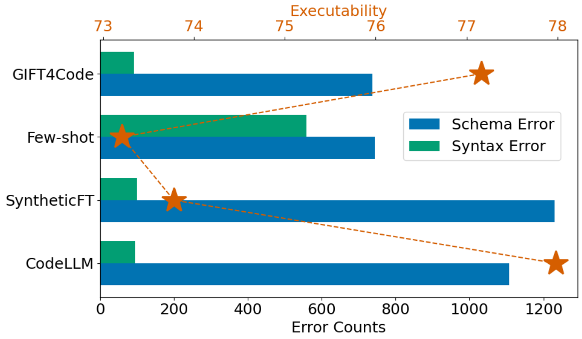


Figure 3: Frequency of error types and code execution rate for different methods. Bottom x-axis stands for the counts of schema errors and syntax errors. Top x-axis represents execution rate. Instruction fine-tuning without specifications (SyntheticFT) decreases executability. Few-shot prompting with specifications reduces schema understanding errors with more syntax errors. GIFT4CODE achieves the best performance by combining their strengths.

as **SyntheticFT** in Fig. 3. The model's execution rate decreases in this scenario because it attempts to better align with user intent, leading to a higher rate of schema understanding errors (*e.g.* referring to non-existing columns as in cell 4, Example 2, Fig. 2).

Incorporating I/O specifications using few-shot prompting leads to another interesting observation. We observe a reduction in schema understanding errors, indicating that the model indeed benefits from the extra specifications. However, given the vast diversity of Python I/O specifications, it is impossible to cover all variants within the few-shot exemplars. Consequently, the model struggles to reliably leverage the specifications, leading to a surge in syntax errors when referring to arbitrarily string-valued schema elements (*e.g.* cell 5, Example 2, Fig. 2). GIFT4CODE effectively mitigates these syntax errors, leading to a higher execution rate while achieving the best *pass*@k accuracies.

5 Related Work

Execution Guided Code Generation One area of study primarily focuses on utilizing execution as I/O examples, facilitating the synthesis of programs that align with the intended behavior. Gulwani (2016) involves synthesizing intended programs in an underlying domain-specific language (DSL) from example based specifications. This method has been further explored and adapted to different applications in subsequent studies (Devlin et al., 2017; Chen et al., 2018; Bunel et al., 2018). Another strand of research (Chen et al., 2021b; Wang et al., 2018; Ellis et al., 2019) leverages intermediate execution results to guide the search of programs. More recently, there have been attempts to utilize program execution results to verify and select code samples predicted by LLMs, either during auto-regressive decoding to prune search space (Zhang et al., 2023), or by few-shot prompting (Chen et al., 2023) and post-hoc reranking (Shi et al., 2022; Ni et al., 2023a). CodeT (Chen et al., 2022) was proposed to generate tests then to conduct dual execution agreements to filter the generated solutions during the inference time.

Instruction Fine-tuning Instruction fine-tuning is a widely adopted approach to address the misalignment issue in LLM-generated content. LLMs such as FLAN (Wei et al., 2021), which excel at understanding and executing instructions from prompts, are trained on labeled training data. Reinforcement learning with human feedback (RLHF) aims to mitigate the amount of labeling effort using model-based reward (Ouyang et al., 2022). Other works also confirmed the effectiveness of using instructional data in the fine-tuning stage (Mishra et al., 2021; Sanh et al., 2021; Chung et al., 2022; Wang et al., 2022b). To lower labeling cost, several recent works explored the possibility of automatic instruction generation (Ye et al., 2022; Zhou et al., 2022; Honovich et al., 2022b). STAR (Zelikman et al., 2022) bootstraps the model's reasoning ability using self-generated rationales. Our work differs from this line by considering execution-based specifications. Additionally, recent works attempted to distill instruction following data from more capable LLMs that have already been instruction-tuned (Honovich et al., 2022a; Taori et al., 2023; Chiang et al., 2023; Peng et al., 2023). In contrast, GIFT4CODE generates synthetic data from vanilla LLMs that have not gone through instruction-tunning.

Synthetic Data from LLMs Besides generating data for instruction following, a number of recent studies have also harnessed general-purpose LLMs to generate realistic synthetic data in areas where labeled data limited, such as language understanding and clinical research (Rosenbaum et al., 2022a; Tang et al., 2023; Borisov et al., 2022; Liu et al., 2022; Rosenbaum et al., 2022b; Josifoski et al., 2023). To improve the quality of synthetic data extracted from LLMs, such approaches usually apply a rejection sampling procedure and filter predictions based on domain-specific heuristics such as logical consistency (Bhagavatula et al., 2022; Yin et al., 2022). GIFT4CODE is in spirit of this line in that it leverages program execution feedback to filter code predictions (Xu et al., 2020).

6 Conclusion

We have presented GIFT4CODE, a framework for instruction fine-tuning large language models of code in which the training is guided by execution based specifications. Empirically, we demonstrated how our approach enhances the quality of generated code by following user-provided specifications, substantially improving accuracy on two challenging data science benchmarks, ARCADE and DS-1000.

References

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *ArXiv*, abs/1611.01989, 2016.

Berant, J., Chou, A., Frostig, R., and Liang, P. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Bhagavatula, C., Hwang, J. D., Downey, D., Bras, R. L., Lu, X., Sakaguchi, K., Swayamdipta, S., West, P., and Choi, Y. I2d2: Inductive knowledge distillation with neurologic and self-imitation. *ArXiv*, abs/2212.09246, 2022.

Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., and Gazit, I. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, jan 2023. ISSN 1542-7730. doi: 10.1145/3582083. URL https://doi.org/10.1145/3582083.

Borisov, V., Sessler, K., Leemann, T., Pawelczyk, M., and Kasneci, G. Language models are realistic tabular data generators. *ArXiv*, abs/2210.06280, 2022.

Bunel, R., Hausknecht, M. J., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *ArXiv*, abs/1805.04276, 2018.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests. *ArXiv*, abs/2207.10397, 2022. URL https://api.semanticscholar.org/CorpusID:250920542.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S. A., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021a.

Chen, X., Liu, C., and Song, D. X. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Chen, X., Song, D. X., and Tian, Y. Latent execution for neural program synthesis beyond domain-specific languages. In *Neural Information Processing Systems*, 2021b.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023.

Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N. M., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B. C., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., García, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Díaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K. S., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Valter, D., Narang, S., Mishra, G., Yu, A. W., Zhao, V., Huang, Y., Dai, A. M., Yu, H., Petrov, S., hsin Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. Scaling instruction-finetuned language models. *ArXiv*, abs/2210.11416, 2022.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., rahman Mohamed, A., and Kohli, P. Robustfill: Neural program learning under noisy i/o. *ArXiv*, abs/1703.07469, 2017.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J. B., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a repl. In *Neural Information Processing Systems*, 2019.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Google. PaLM2 technical report. `https://ai.google/static/documents/palm2techreport.pdf`, 2023. Accessed: 2023-05-10.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 2011. URL `https://api.semanticscholar.org/CorpusID:886323`.

Gulwani, S. Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, 2016.

Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM*, 55: 97–105, 2012.

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

Honovich, O., Scialom, T., Levy, O., and Schick, T. Unnatural instructions: Tuning language models with (almost) no human labor. *ArXiv*, abs/2212.09689, 2022a.

Honovich, O., Shaham, U., Bowman, S. R., and Levy, O. Instruction induction: From few examples to natural language task descriptions. *ArXiv*, abs/2205.10782, 2022b.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Mapping language to code in programmatic context. *ArXiv*, abs/1808.09588, 2018.

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1219–1231, 2022.

Josifoski, M., Sakota, M., Peyrard, M., and West, R. Exploiting asymmetry for synthetic training data generation: Synthie and the case of information extraction. *ArXiv*, abs/2303.04132, 2023.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S. M., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you! 2023.

Li, Y., Choi, D. H., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Tom, Eccles, Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de, C., d'Autume, M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Alexey, Cherepanov, Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de, N., Freitas, Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097, 2022.

Liu, Q., Ye, Z., Yu, T., Blunsom, P., and Song, L. Augmenting multi-turn text-to-sql datasets with self-play. In *Conference on Empirical Methods in Natural Language Processing*, 2022.

Mishra, S., Khashabi, D., Baral, C., and Hajishirzi, H. Cross-task generalization via natural language crowdsourcing instructions. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

Ni, A., Iyer, S., Radev, D. R., Stoyanov, V., tau Yih, W., Wang, S. I., and Lin, X. V. Lever: Learning to verify language-to-code generation with execution. *ArXiv*, abs/2302.08468, 2023a.

Ni, A., Yin, P., Zhao, Y., Riddell, M., Feng, T., Shen, R., Yin, S., Liu, Y., Yavuz, S., Xiong, C., Joty, S. R., Zhou, Y., Radev, D. R., and Cohan, A. L2ceval: Evaluating language-to-code generation capabilities of large language models. *ArXiv*, abs/2309.17446, 2023b. URL `https://api.semanticscholar.org/CorpusID:263310373`.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. 2022.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller,

L. E., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. J. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

Peng, B., Li, C., He, P., Galley, M., and Gao, J. Instruction tuning with gpt-4. *ArXiv*, abs/2304.03277, 2023.

Rosenbaum, A., Soltan, S., Hamza, W., Saffari, A., Damonte, M., and Groves, I. Clasp: Few-shot cross-lingual data augmentation for semantic parsing. In *AACL*, 2022a.

Rosenbaum, A., Soltan, S., Hamza, W., Versley, Y., and Boese, M. Linguist: Language model instruction tuning to generate annotated utterances for intent classification and slot tagging. In *International Conference on Computational Linguistics*, 2022b.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M. P., Ferrer, C. C., Grattafiori, A., Xiong, W., D'efossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. URL `https://api.semanticscholar.org/CorpusID:261100919`.

Sanh, V., Webson, A., Raffel, C., Bach, S. H., Sutawika, L., Alyafeai, Z., Chaffin, A., Stiegler, A., Scao, T. L., Raja, A., Dey, M., Bari, M. S., Xu, C., Thakker, U., Sharma, S., Szczechla, E., Kim, T., Chhablani, G., Nayak, N. V., Datta, D., Chang, J., Jiang, M. T.-J., Wang, H., Manica, M., Shen, S., Yong, Z. X., Pandey, H., Bawden, R., Wang, T., Neeraj, T., Rozen, J., Sharma, A., Santilli, A., Févry, T., Fries, J. A., Teehan, R., Biderman, S. R., Gao, L., Bers, T., Wolf, T., and Rush, A. M. Multitask prompted training enables zero-shot task generalization. *ArXiv*, abs/2110.08207, 2021.

Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., and Wang, S. I. Natural language to code translation with execution. *ArXiv*, abs/2204.11454, 2022.

Shi, K., Bieber, D., and Singh, R. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44:1 – 36, 2020. URL `https://api.semanticscholar.org/CorpusID:214605958`.

Tang, R., Han, X., Jiang, X., and Hu, X. Does synthetic data generation of llms help clinical text mining? *ArXiv*, abs/2303.04360, 2023.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P.-S., Mao, Y., Polozov, O., and Singh, R. Robust text-to-sql generation with execution-guided decoding. *arXiv: Computation and Language*, 2018.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning language model with self generated instructions. *ArXiv*, abs/2212.10560, 2022a.

Wang, Y., Mishra, S., Alipoormolabashi, P., Kordi, Y., Mirzaei, A., Arunkumar, A., Ashok, A., Dhanasekaran, A. S., Naik, A., Stap, D., Pathak, E., Karamanolakis, G., Lai, H. G., Purohit, I., Mondal, I., Anderson, J., Kuznia, K., Doshi, K., Patel, M., Pal, K. K., Moradshahi, M., Parmar, M., Purohit, M., Varshney, N., Kaza, P. R., Verma, P., Puri, R. S., Karia, R., Sampat, S. K., Doshi, S., Mishra, S. D., Reddy, S., Patro, S., Dixit, T., Shen, X., Baral, C., Choi, Y., Smith, N. A., Hajishirzi, H., and Khashabi, D. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Conference on Empirical Methods in Natural Language Processing*, 2022b.

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners. *ArXiv*, abs/2109.01652, 2021.

Xu, S., Semnani, S. J., Campagna, G., and Lam, M. S. Autoqa: From databases to q&a semantic parsers with only synthetic training data. *ArXiv*, abs/2010.04806, 2020.

Ye, S., Kim, D., Jang, J., Shin, J., and Seo, M. Guess the instruction! flipped learning makes language models stronger zero-shot learners. *ArXiv*, abs/2210.02969, 2022.

Yin, P., Wieting, J. F., Sil, A., and Neubig, G. On the ingredients of an effective zero-shot semantic parser. In *Annual Conference of the Association for Computational Linguistics (ACL)*, Dublin, Ireland, May 2022. URL `https://arxiv.org/abs/2110.08381`.

Zelikman, E., Huang, Q., Poesia, G., Goodman, N. D., and Haber, N. Parsel: A (de-)compositional framework for algorithmic reasoning with language models. 2022.

Zhang, S., Chen, Z., Shen, Y., Ding, M., Tenenbaum, J. B., and Gan, C. Planning with large language models for code generation. *ArXiv*, abs/2303.05510, 2023.

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers. *ArXiv*, abs/2211.01910, 2022.

A Applying GIFT4CODE to Data Science Code Generation

In this appendix section, we demonstrate the practical application of our proposed approach GIFT4CODE, as discussed in §3, to a specific dataset, ARCADE (Yin et al., 2022). We follow the same setup as described in §3. Our starting point involves a "Generalist LLM" (PALM2) and a Code LLM (PaLM based 62B model in our case), the alignment of which we seek to improve.

A.1 Synthetic Data Generation

We first gathered CSV files from diverse GitHub repositories focusing on data science projects. These CSV files encompass a wide range of data types, structures, and domains, serving as the programmatic context to initialize the synthetic data generation. Subsequently, we employed a "Generalist" PALM2 LLM specifically designed for natural language understanding tasks, distinguished from the base code LLMs. This model was utilized to generate natural language questions based on the information extracted from the collected CSV files. For the following sections, we denote the set of natural number from 0 to W and $\mathbb{N}_W = \{1, 2, 3, \ldots, W\}$.

Creating NL Intents (Questions) Using the entire CSV file as the programmatic context to prompt the "Generalist" PALM2 LLM is infeasible due to its length after tokenization. Instead, we extracted the header for each CSV file as the programmatic context $\boldsymbol{c}_i$ to query the LLM to generate the NL intents $\{\boldsymbol{x}_i\}$. Given the length of an entire CSV file after tokenization, using it as the programmatic context to prompt the "generalist" LLM is impractical. Instead, we derive the header and its first three rows from each CSV file as the programmatic context, denoted as $\boldsymbol{c}_i$, to query the LLM to produce NL intents $\boldsymbol{x}_i$. To construct the few-shot exemplars, we randomly selected a subset of CSV headers, denoted as $\boldsymbol{C}' = \{\boldsymbol{c}_i' \mid i \in \mathbb{N}_P\}$ where $P < 10$. The prime symbol $'$ in the superscript denotes the variable will be used in the few-shot exemplars. We manually crafted a set of 10 data science-related questions corresponding to each $\boldsymbol{c}_i'$, denoted as $\boldsymbol{X}_i' = \{\boldsymbol{x}_i^j \mid j \in \mathbb{N}_{10}\}$. This process allowed us to create a few-shot prompt set $\mathcal{P}$ that consists of pairs of CSV headers and associated questions, formulated as $\mathcal{P} = \{(\boldsymbol{c}_i', \boldsymbol{X}_i')\}_{i=1}^P$.

We employed the standard few-shot prompting approach, which concatenates $\mathcal{P}$ to each CSV header $\boldsymbol{c}_i$ when querying the LLM. In specific, each header $\boldsymbol{c}_i$ is augmented to $\hat{\boldsymbol{c}}_i = (\mathcal{P}, \boldsymbol{c}_i)$. In this configuration, the LLM is prompted to generate questions related to $\boldsymbol{c}_i$, while emulating the style of $\mathcal{P}$. After generation, we perform diversity filtering on $\boldsymbol{X}_i$. Let $\widehat{\boldsymbol{X}_i}$ denote the set of NL questions after the filtering process. We initialize $\widehat{\boldsymbol{X}_i}$ as $\{q_0\}$ where $q_0 \sim \mathcal{U}(\boldsymbol{X}_i)$), a question randomly selected from $\boldsymbol{X}_i$. Similar to Wang et al. (2022a), we iterate over $\{\boldsymbol{X}_i \setminus q_0\}$, a new question is added to the set $\widehat{\boldsymbol{X}_i}$ only when its ROUGE-L overlap with any $q \in \widehat{\boldsymbol{X}_i}$ is less than 0.7. Questions that are repetitive are also filtered out. For notation simplicity, in the following sections, we use $\boldsymbol{X}_i$ to represent the set of NL intents after the filtering process. Empirically, we observe that the NL questions generated by this LLM encompassed a wide range of tasks relevant to the given programmatic context, as shown in Listing 2 and Listing 3.

Predicting Code Solutions Subsequently, we employed the code LLM, which we aim to improve, to generate code that addresses these NL questions. For each question $q_i^j \in \boldsymbol{X}_i$ where j denotes the jth question associated with the programmatic context $\boldsymbol{c}_i$, we can draw code solutions from $P_{\text{LLM}}(\boldsymbol{y} \mid \boldsymbol{c}_i, q_i^j)$. This leads to a pool of code samples $\{\boldsymbol{y}\}_i^j$. Following this, we applied heuristic filtering on these code samples, adopting two criteria: 1) Maximizing the diversity of Python library API calls, and 2) Ensuring the produced execution output is meaningful, for instance, yielding a `pandas.DataFrame` object if the code is executable. This process resulted in a synthetic dataset $\mathcal{D}$ that enables instruction fine-tuning. Notice that $\mathcal{D}$ does not contain any I/O specifications yet.

The synthetic data generation process leverages the capabilities of the LLM to create a diverse set of natural language intents that correspond to the header information extracted from the CSV files. By incorporating the knowledge and understanding of "Generalist" PALM2 LLM, we are able to generate a rich and varied synthetic dataset that contains a wide range of data science tasks and scenarios.

Executing Code Samples Executing code derived from a pre-training corpus which was used to train the code LLM can be challenging, as it often demands notoriously complicated environment setup. In contrast, the synthetic code samples offer a significant advantage, as they can be executed more easily without the need for complex environment setup. Moreover, the execution of synthetic code samples enables us to track variables and their states both before and after running the code. This information can be utilized to identify input variables that exhibit state changes after execution. We label these variables as inputs to the code. In addition, by examining the final states of these input variables, we can effectively establish the input-output specifications of the code.

B LLM Generated NL Intents

In this section, we demonstrated the NL questions generated by the "generalist" PALM2 LLM on the ARCADE dataset. To begin, we provide an exemplar used in the few-shot prompt when querying the LLM to generate NL questions, as seen in Listing 1. The given exemplar consists of the CSV header along with the first three rows. If any entry within the first three rows exceeds 50 characters in its string representation, we truncate it to maintain conciseness. As shown in Listing 1, we handcrafted 10 diverse questions, covering as many data science topics as possible.

In this section, we provide two examples of NL intent generation. Each of these examples, as shown in Listing 2 and Listing 3, includes both the programmatic context and the output generated by the LLM. Listing 2 illustrates an instance regarding a Pokémon game experience. Notably, the LLM tends to generate relatively basic questions in this context, which don't necessitate the use of complex Python APIs such as `pandas.groupby`. Conversely, Listing 3 presents an example related to a Netflix TV and movie dataset. In this case, the LLM produces more comprehensive questions. Addressing these questions requires multiple API calls, indicating a higher level of complexity.

C I/O Summary Examples

To begin with, we showed an example of a few-shot exemplar used to prompt the "generalist" PALM2 LLM in generating an I/O summary for the ARCADE dataset, as detailed in section 3.2. The exemplar in Listing 4 comprises an input dataframe schema, a Python code solution, execution output, and user intent. The anticipated response to this prompt is an I/O summary, outlining the input-output variable names and their types. In this example, there is only one variable — ""alc"" which is a `pandas.DataFrame`. Next, the LLM is expected to give a succinct description on the salient input columns, as well as a brief summary of the example output columns.

We present two examples of LLM generated I/O summaries on the ARCADE dataset, as illustrated in Listing 5 and Listing 6. As mentioned in §4.1, we deliberately obscure the execution output details when prompting the LLM to generate an I/O summary. This step helps to more realistically simulate user provided specifications. Each example consists of its prompt we used to query the LLM for the I/O summary, the resulting example augmented by the generated I/O summary and the ground truth execution output which was never exposed to LLM.

The first example Listing 5 focuses on a dataset detailing mobile phone information, with the user intent being to determine the quantity of different smartphones released each decade. The subsequent I/O summary generated by the LLM identifies the "smartphone" and "year" columns as the most relevant from the input dataframe and describes the output values as being of int64 type, with an index year. The generated I/O summary correctly describes the ground truth presented in the bottom of Listing 5.

In the second example Listing 6, the LLM is dealing with movie data where the user's intent is to chronologically list the worldwide gross of films released each month from November 2021 to June 2022. The corresponding I/O summary generated by the LLM correctly identified the salient columns in the input dataframe, "Worldwide" and "Released Date". However, an observed discrepancy between the ground truth output and the I/O summary indicates that the LLM's generation is not entirely accurate for this specific example.

These two test examples on ARCADE illustrate the LLM's capabilities in generating I/O summaries, while highlighting the potential discrepancies that may occur.

D Additional Experimental Results

In this section, we offer two additional experiments that supplement the results presented in §4.2. We have demonstrated the GIFT4CODE model consistently outperforms in all types of I/O specifications. A natural follow-up question might be whether the instruction fine-tuning degrades the model's programming ability if no specifications are provided by the user. To address this concern, we have conducted further experiments where an instruction tuned model is evaluated on the ARCADE dataset, in the absence of any specification.

The results in Tab. 3 demonstrate that while the instruction tuned model does perform slightly worse than the model fine-tuned on data with no specifications, the difference is marginal. In the *pass*@5 and *pass*@20 settings, both with no

```
First 3 rows from dataset 4-wheeler-EV.csv (column data types in parentheses)
| Comfort (float) | Performance (float) | Fuel Economy (float)
| Value for Money (float) | Condition (string) | driven (string) | rating (float)
| model_name (string) |
|----------------------------------------------------------------------------------|
| 4 | 5 | 5 | 5 | New | Few hundred kilometers | 5 | hyundai kona |
| 1 | 1 | 1 | 1 | New | Haven’t driven it | 0 | hyundai kona |
| 5 | 5 | 5 | 4 | New | Few thousand kilometers | 5 | hyundai kona |

First 3 rows from dataset 2-wheeler-EV.csv (column data types in parentheses)
| Owned for (string) | Ridden for (string) | rating (int)
| Visual Appeal (float) | Reliability (float) | Performance (float)
| Service Experience (float) | Extra Features (float) | Comfort (float)
| Maintenance cost (float) | Value for Money (float) | Model Name (string) |
|----------------------------------------------------------------------------------|
| Never owned | nan | 1 | 3 | 4 | nan | nan | nan | 4 | nan | 1 | TVS iQube |
| > 1 yr | < 5000 kms | 1 | 3 | 1 | nan | 1 | nan | 3 | nan | 3 | TVS iQube |
| < 3 months | < 5000 kms | 3 | 4 | 4 | nan | 2 | nan | 5 | nan | 2 | TVS iQube |

Here are a series of contextually dependent data wrangling and exploratory data
analysis tasks for the dataset:
Task 1: How many new cars have a performance rating of more than 3?
Task 2: Among the Hyundai Kona reviews, do those with a purchase history have
average performance rating better than the ones that have not yet been purchased?
Task 3: Create a new column called "Review’s Word Count" and Show the rating of
the shortest review
Task 4: Which model has the highest average word count?
Task 5: How many reviews are for used cars for this model?
Task 6: What are the top five models with most number of bikes having mileage
less than 5000 kilometers
Task 7: Which of these models has the highest comfort score on average?
Task 8: What are those models mostly used for?
Task 9: How many times the word scooter was mentioned in the bike’s reviews?
Task 10: Show a list of words that were mentioned only once in the reviews
[END]
```

Listing 1: An exemplar in the few-shot prompt when querying LLM to generate NL intents.

```
| id (int) | identifier (string) | species_id (int) | height (int) | weight (int)
| base_experience (int) | order (int) | is_default (int) |
|-----------------------------------------------------------------------------|
| 1 | bulbasaur | 1 | 7 | 69 | 64 | 1 | 1 |
| 2 | ivysaur | 2 | 10 | 130 | 142 | 2 | 1 |
| 3 | venusaur | 3 | 20 | 1000 | 263 | 3 | 1 |
```

(a) The programmatic context to query the “generalist” PALM2 LLM, albeit only a subset of the columns are displayed.

```
Here are a series of contextually dependent data wrangling and exploratory data
analysis tasks for the dataset:
Task 1: How many pokemons are there in the Pokedex?
Task 2: Find the total number of columns that are integers.
Task 3: Calculate the ratio of mean weight to height for each pokemon
Task 4: What is the weight of “Snorlax” ?
Task 5: How many pokemon have the same average base experience as their id?
Task 6: Find the order for each pokemon that weighs less than 100 pounds
Task 7: What is the “mean” and “median” of “height” column ?
Task 8: Show the names of the pokemons with minimum and maximum weight, height
and base experience.
Task 9: Show the first 20 and last 10 pokemon with their average base experience.
Task 10: Create a new column called "size_cat" that has size categories for
pokemon (child: 1-10, teenager: 11-20, adult: 21+)
```

(b) LLM generated questions for the above programmatic context.

Listing 2: Programmatic context and generated questions on an example concerning pokemon game experience dataset.

context and full context, the model's performance (all types of specifications) only decreases by most 2% when compared with the model without specifications. This is expected as the discrepancy between the fine-tuning data and the testing data could lead to a minor regression in performance. This marginal decrease in performance is counterbalanced by the significant improvement we previously observed in §4.2. Therefore, GIFT4CODE with I/O summary still remains the superior method for instruction fine-tuning.

```
| show_id (string) | type (string) | title (string) | director (string) |
| country (string) | date_added (string) | release_year (int) | ... |
|-----------------------------------------------------------------------------|
| s1 | Movie | Dick Johnson Is Dead | Kirsten Johnson | United States |
    September 25, 2021 | 2020 | PG-13 | 90 min | ... |
| s2 | TV Show | Blood & Water | nan | South Africa |
    September 24, 2021 | 2021 | TV-MA | 2 Seasons | ... |
| s3 | TV Show | Ganglands | Julien Leclercq | nan |
    September 24, 2021 | 2021 | TV-MA | 1 Season | ... |
```

(a) The programmatic context to query the "generalist" PaLM2 LLM, albeit only a subset of the columns are displayed.

```
Here are a series of contextually dependent data wrangling and exploratory data
analysis tasks for the dataset:
Task 1: What is the count of number of seasons for each show?
Task 2: How Many shows have “TV14” as rating?
Task 3: How many movies have a rating of 13+?
Task 4: Show the top 10 TV shows with most number of seasons
Task 5: Make a new column "genre" that combines all genres into one column.
Task 6: Show the names of casts who have been in at least 5 shows?
Task 7: How many TV Shows are there that have been released before 2017?
Task 8: For each director, how many shows have been added in 2020?
Task 9: Show the movies that have director’s name with the letter ’b’ in it.
Task 10: Show the number of shows released before 2020 in the genre “Documentaries”?
Task 11: For each type, what are the average durations?
(Show dataframe that has type and duration as columns)
```

(b) LLM generated questions for the above programmatic context.

Listing 3: Programmatic context and generated questions on an example concerning Netflix TV show and movie dataset.

```
"""
The input dataframe schema is:
Schema of Dataframes:
Columns in alc with example values:
country (Afghanistan), beer_servings (0), spirit_servings (0), wine_servings (0),
total_litres_of_pure_alcohol (0.0), continent (AS)
"""

# The Python solution is:
import pandas as pd
alc = pd.read_csv("drinks.csv")
alc['continent'].unique()
alc.groupby('continent').agg({'beer_servings': np.mean}).sort_values(
    'beer_servings', ascending=True)
alc.dropna()

# The execution output is:
alc:
|  beer_servings (float) |
|------------------------|
|   102   |
|    20   |

# The user intent is:
Rank the continents that consume the least amount of beer on average.

# The I/O specification is:
alc: a pandas.core.DataFrame. Given the user intent and the code, the salient
columns (at most given 3) in the input dataframe are beer_servings, continent.
The output dataframe has columns (at most given 3) such as beer_servings.
```

Listing 4: An exemplar regarding `pandas.DataFrame`.

```
"""
The input schema is:
# Schema of Dataframes:
# Columns in phones with example values:
# manufacturer (Nokia), model (1100), form (Bar), smartphone (No), year (2003)
"""

# The Python solution is:
yearly_smartphones = phones.groupby(['year', 'smartphone'],
    as_index=False).size().pivot_table(
        index='year',columns='smartphone', values='size').fillna(0)
yearly_smartphones.groupby((yearly_smartphones.index//10)*10).Yes.sum()

# The execution output is:
__output__:
pandas.core.series.Series

# The user intent is:
How many different smartphones were released each decade?

The I/O specification is:
```

(a) The prompt used to query LLM for I/O summary.

```
# In[ ]:
import pandas as pd
import numpy as np
phones = pd.read_csv('dataset/best-selling-mobile-phones.csv')
# In[ ]:
# Schema of Dataframes:
# Columns in phones with example values:
# manufacturer (Nokia), model (1100), form (Bar), smartphone (No), year (2003)
# In[ ]:
phones[phones.form.str.lower().str.contains('touchscreen')].groupby(
    'manufacturer').model.nunique().idxmax()
# In[ ]:
year_phones = phones[phones['year'] >= phones['year'].max()-15]
year_phones.groupby(['year','manufacturer','form'], as_index=False).size().pivot(
    index=['year','manufacturer'], columns='form').fillna(0)

# In[ ]:
How many different smartphones were released each decade?
"""
Input-output Summary:
__output__: a pandas.core.series.Series. Given the user intent and the code, the
salient columns (at most given 3) in the input dataframe are smartphone, year.
The output values are of type int64, with an index year. Here is my code solution:
"""
# In[ ]:
```

(b) A test example on ARCADE augmented with a LLM generated I/O summary.

```
year
1990     0.0
2000    12.0
2010    58.0
Name: Yes, dtype: float64
```

(c) The ground truth of the output in the above test example.

Listing 5: An example of an I/O summary generated by the LLM on ARCADE for a `pandas.Series`.

```
"""
The input schema is:
Schema of Dataframes:
Columns in df with example values:
Movie (JugJugg Jeeyo), Worldwide (74.5), India Net (50.24), India Gross (54.5),
Budget (100), Verdict (None), Movie Type (Bollywood), Released Date (24-Jun-22)
"""

# The Python solution is:
df_t[['month', 'Worldwide']].groupby('month').sum().T

# The execution output is:
__output__:
pandas.core.frame.DataFrame

# The user intent is:
List the worldwide gross of the films released for each month since November,
2021 to June 2022. List  the months in chronological order.

# The I/O specification is:
```

(a) The prompt used to query LLM for I/O summary.

```
# In[ ]:
from datetime import datetime
import pandas as pd
df = pd.read_csv('dataset/bollywood_2022.csv')
# In[ ]:
# Schema of Dataframes:
# Columns in df with example values:
# Movie (JugJugg Jeeyo), Worldwide (74.5), India Net (50.24), India Gross (54.5),
# Overseas (20.0), Movie Type (Bollywood), Released Date (24-Jun-22)
# In[ ]:
df.columns = [column.replace(' ', '') for column in df.columns]

# In[ ]:
List the worldwide gross of the films released for each month # since
November, 2021 to June 2022. List the months in chronological order.
"""
Input-output Summary:
__output__: a pandas.core.DataFrame. Given the user intent and the code,
the salient columns (at most given 3) in the input dataframe are Worldwide,
Released Date. The output dataframe has columns (at most given 3) such as month,
Worldwide. Here is my code solution:
"""
# In[ ]:
```

(b) A test example on ARCADE augmented with a LLM generated I/O summary.

```
| April, 2022 (float) | December, 2021 (float) | February, 2022 (float)
| January, 2022 (float) | June, 2022 (float) | March, 2022 (float)
| May, 2022 (float) | November, 2021 (float) |
|-------------------------------------------------------------|
| 1748.13 | 10988.6 | 1812.99 | 114.12 | 4763.05 | 3849.25 | 7730.75 | 169.23 |
```

(c) The ground truth dataframe of the output in the above test example.

Listing 6: An noisy example of LLM generated I/O summary. The LLM generated I/O specification is inaccurate as evident from the discrepancies between the I/O summary in part **(b)** and the ground truth in part **(c)**.

Methods	ARCADE			
	pass@5		*pass*@20	
	No Context	Full Context	No Context	Full Context
GIFT4CODE				
Code LLM (no spec.)	**20.78**	**34.33**	**33.40**	**46.94**
+ TypeDesc	20.59	33.43	32.98	46.01
+ I/O Examples	20.04	31.76	31.40	43.60
+ I/O Summary	20.04	32.39	32.24	45.14

Table 3: *pass*@k on ARCADE. For each type of I/O specification in Tab. 3 (*e.g.* +I/O Summary), intents are augmented with I/O specifications of that type (*e.g.* intents inline with I/O summary) in fine-tuning data. At test time, input intents do not contain have any specification.