Table-LLM-Specialist: Language Model Specialists for Tables using Iterative Fine-tuning

Junjie Xing[*]
University of Michigan

Yeye He[†]**, Mengyu Zhou, Haoyu Dong, Shi Han, Dongmei Zhang, Surajit Chaudhuri**
Microsoft Corporation

Abstract

Language models such as GPT and Llama have shown remarkable ability on diverse natural language tasks, yet their performance on complex table tasks (e.g., NL-to-Code, data cleaning, etc.) continues to be suboptimal. To improve their performance, task-specific fine-tuning is often needed, which, however, require expensive human labeling and is prone to over-fitting.

In this work, we propose TABLE-SPECIALIST, a self-trained fine-tuning paradigm specifically designed for table tasks. Our insight is that for each table task, there often exist two dual versions of the same task, one *generative* and one *classification* in nature. Leveraging their duality, we propose a *Generator-Validator* paradigm to iteratively generate-then-validate training data from language models, to fine-tune stronger TABLE-SPECIALIST models that can specialize in a given task, without using manually-labeled data.

Extensive evaluations of TABLE-SPECIALIST on Llama, GPT-3.5 and GPT-4 suggest that our TABLE-SPECIALIST has (1) *strong performance* on diverse tasks over vanilla language-models – for example, TABLE-SPECIALIST fine-tuned on GPT-3.5 not only outperforms vanilla GPT-3.5, but can often surpass GPT-4 level quality, (2) *lower cost* to deploy, because when TABLE-SPECIALIST fine-tuned on GPT-3.5 achieve GPT-4 level quality, it becomes possible to deploy smaller models with lower latency/cost at comparable quality, and (3) *better generalizability* when evaluated across multiple benchmarks, since TABLE-SPECIALIST is fine-tuned on a broad range of training data systematically generated from diverse real tables.

Our code is available at microsoft/Table-Specialist. Specialist models fine-tuned using TABLE-SPECIALIST have been integrated into Microsoft Excel for use cases such as automated data cleaning.

1 Introduction

Language models, such as GPT (Brown et al., 2020) and Llama (Touvron et al., 2023), have shown remarkable abilities to perform diverse natural language tasks with strong generalizability.

However, when it comes to complex "table tasks", such as data transformation (He et al., 2018; Harris and Gulwani, 2011), data cleaning (Rahm et al., 2000; Wang and He, 2019; Chen et al., 2025), column type annotation (Korini and Bizer, 2023; Hulsebos et al., 2019), where the central object of interest is a *structured table* (as opposed to natural language text), even the latest language models can struggle to perform well (Li et al., 2024b; Zhang et al., 2024b; Tian et al., 2024), likely because language models are pre-trained predominately on one-dimensional text, whereas tables are two-dimensional objects (Li et al., 2024b; Sui et al., 2024; Tian et al., 2024; Xing et al., 2025).

Prior approaches: Fine-tuning for table tasks. To improve the performance of language models on table tasks, different fine-tuning techniques have been used, which we summarize as follows:

Dataset-specific fine-tuning. A straightforward approach is what we call "dataset-specific fine-tuning". Given a particular table-task T (say data-transformation), we start from a base model M such as GPT or Llama, and use the training split of a labeled dataset D to fine-tune M, which can often lead to significant gains on the (highly-similar) test-split of D, which is a common approach used in many benchmarks for table-tasks (Yu et al., 2018; Zhong et al., 2017; Das et al.; em-; Zhang et al., 2024c; Pasupat and Liang, 2015).

However, as one may expect, language models fine-tuned on the training-split of one dataset D often do not generalize well to another dataset D' for the same task type T. For instance, we find that the models fine-tuned for NL-to-SQL using the Wiki-SQL (Zhong et al., 2017) dataset does

[*] Email: jjxing@umich.edu, work done at Microsoft
[†] Correspondence to: yeyehe@microsoft.com.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 35432–35449
November 4-9, 2025 ©2025 Association for Computational Linguistics

not generalize to different NL-2-SQL datasets like Spider (Yu et al., 2018) or BIRD (Li et al., 2024a). Because the relatively narrow nature of one specific dataset, often manually labeled at a small scale, can lead to poor "generalizability" when using "dataset-specific fine-tuning" [1].

Table-Generalist fine-tuning. A second class of table fine-tuning techniques are inspired by general-purpose chat models like ChatGPT and Llama-Chat, which are fine-tuned from base models (GPT and Llama, respectively) (Ouyang et al., 2022; Wang et al., 2022b; cha) for instruction-following, with great generalizability to handle diverse human instructions, which we term "Chat-Generalist".

Inspired by the success of "Chat-Generalist", "Table-Generalist" models like Table-GPT (Li et al., 2024b) and Table-Llama (Zhang et al., 2024b) are developed, which are fine-tuned similarly to ChatGPT by pooling diverse table-tasks as training data for multi-task table fine-tuning. The resulting "Table-Generalist" models can handle diverse table-tasks, with better performance on a wide range of table tasks than the vanilla GPT/Llama, including on *new and unseen table-tasks* held out during fine-tuning (Li et al., 2024b; Zhang et al., 2024b).

However, as one may expect, their cross-task generality comes at a cost of performance, as there is often a performance gap between "dataset-specific fine-tuning" and Table-Generalists (Li et al., 2024b; Zhang et al., 2024b).

TABLE-SPECIALIST: a new approach to table fine-tuning. In this work, we develop a new fine-tuning approach for table-tasks that aims to close the performance gap, which we call "TABLE-SPECIALIST". In this approach, each TABLE-SPECIALIST model is fine-tuned by design to *focus on one specific type of table task T* (e.g., one model for data transformation, one model for error detection, one model for NL-to-SQL, etc.), which is unlike Table-Generalists (Table-GPT and Table-Llama) that can handle all types of table tasks.

Importantly, by being specialized in one task T, TABLE-SPECIALIST can be (1) made much more performant than Table-Generalists, while (2) still generalize to new and unseen datasets of the same task T (unlike "dataset-specific fine-tuning").

At a high level, our TABLE-SPECIALIST exploits *a duality of table tasks*, where a "*generative table-task*" has a counterpart that is a "*classification table-task*", and vice versa, forming two dual versions of the same task. Correspondingly, we propose a "Generator-Validator" framework that can iteratively fine-tune a generative model and a classification model for the dual versions of the task, using training data automatically "generated-then-validated" by the two models, leveraging unique characteristics of tables (e.g., permutation-invariance and execution-invariance).

While dual-learning and dual-tasks is studied for machine-translation tasks (e.g., translating from language A to B, and from B to A) (Wang et al., 2019; Sennrich and Zhang, 2019), they remain largely unexplored in the context of tables (Section 2). In our work, we introduce novel dual-learning techniques specifically designed for table-tasks, leveraging the unique two-dimensional structure of tables, which are much more generalizable than "dataset-specific fine-tuning", and much more performant than "Table-Generalist" (Section 3).

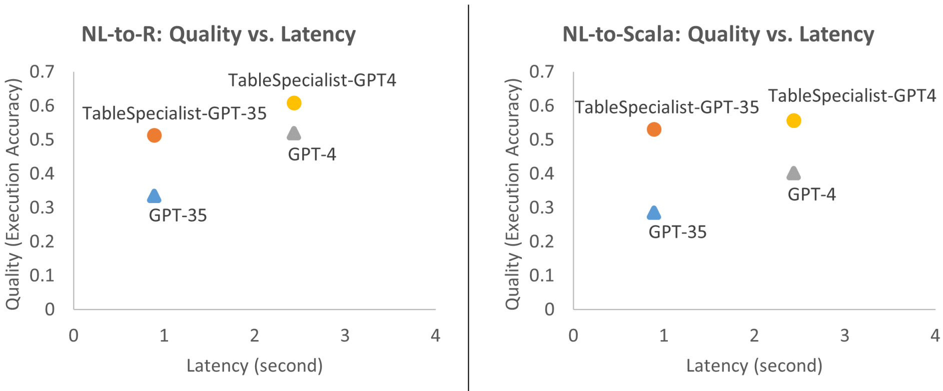
Key benefits of TABLE-SPECIALIST. To better illustrate the benefits of TABLE-SPECIALIST, in Figure 1 we highlight our results on two table-tasks, NL-to-R and NL-to-Scala (which are similar to NL-to-SQL but translate natural language questions to R and Spark-Scala instead). In both figures, we can see that vanilla GPT-4 produces higher quality than vanilla GPT-3.5, but has 2-3x higher latency (and also higher financial cost to deploy). The proposed TABLE-SPECIALIST fine-tuned on GPT-3.5 and GPT-4 show strong quality gains over vanilla GPT-3.5 and GPT-4, respectively, *without* using any training data from the training-split of any benchmarks.

More importantly, we can see that in both cases, TABLE-SPECIALIST-GPT-3.5 can match or exceed the quality achieved by vanilla GPT-4. Furthermore, because these TABLE-SPECIALIST-GPT-3.5 are fine-tuned on GPT-3.5, they have similar latency as vanilla GPT-3.5 (shown on x-axis). What this means is that for these table-tasks, we can deploy smaller specialized models (TABLE-SPECIALIST-GPT-3.5) over larger general models (vanilla GPT-4), with comparable quality, but *at significantly lower latency and costs*.

Key benefits of our proposed approach include:

- Strong performance. TABLE-SPECIALIST outperforms vanilla language models, as well as

[1] While "over-fitting" is a well-known topic (Srivastava et al., 2014; Zhang et al., 2018) especially for small models, our experience reported in more detail in (Xing et al., 2024) shows that over-fitting can still happen on table tasks, even when fine-tuned large language models have a large capacity that are supposed to be robust to over-fitting).



(a) NL-to-R

(b) NL-to-Scala

Figure 1: "TABLE-SPECIALIST fine-tuning": Quality vs. latency comparison on two sample table-tasks: (a) NL-to-R; (b) NL-to-Scala. In both cases, TABLE-SPECIALIST-GPT-3.5 significantly outperforms vanilla GPT-3.5, and even vanilla GPT-4 (shown on y-axis), making it possible to deploy TABLE-SPECIALIST-GPT-3.5 over vanilla GPT-4, at much lower latency and costs (x-axis).

Table-Generalist models.

- Lower cost. Because TABLE-SPECIALIST achieves GPT-4 quality using fine-tuned GPT-3.5 models, it is substantially cheaper to deploy.
- Better generalizability. TABLE-SPECIALIST reliably generalizes to new and unseen datasets for the same task and is benchmark-agnostic.
- Labeling-free. Because the TABLE-SPECIALIST leverages language-models and "Generator-Validator" to automatically produce training data for fine-tuning, it is easier to scale to new table-tasks without expensive human labeling.

2 Related work

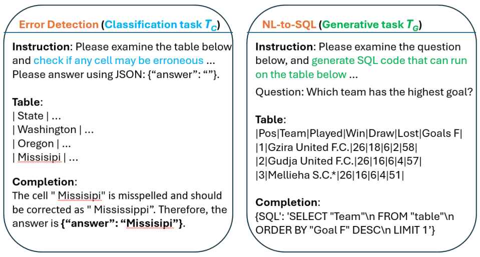
Language models for table tasks. Many tasks studied in the literature center around tables, which are increasingly important (e.g., in database and spreadsheet copilot/assistant scenarios). We study a sample of common table tasks in this work, and refer readers to surveys like (Dong et al., 2022; Dong and Wang, 2024; Xing et al., 2025) for a more comprehensive review of table tasks.

Language models, such as GPT and Llama, are capable of performing not only natural language tasks, but also table tasks (Narayan et al., 2022; Fernandez et al., 2023). However, language models still struggle with complex table tasks (Li et al., 2024b; Zhang et al., 2024b; Sui et al., 2024). This can be attributed to factors such as large table context (Sui et al., 2024; Tian et al., 2024), two-dimensional reasoning (Li et al., 2024b; Sui et al., 2024, 2023), etc., which leads to the need for fine-tuning, as discussed in the introduction.

Train language models using synthetic data. In this work, we fine-tune models for individual table tasks, using synthetic training data "generated-then-validated" by language models from diverse real tables, which is inspired by the success of using synthetic data to train state-of-the-art small language models (Li et al., 2023; Adler et al., 2024; Mukherjee et al., 2023) and text-embedding models (Wang et al., 2023), that are also trained using synthetic data generated by language models. In TABLE-SPECIALIST, we leverage the duality of table tasks and other unique characteristics of tables (e.g., permutation-invariance, and execution-invariance), which are all specific to table tasks.

Validation in table-tasks vs. NLP reasoning tasks. In our Generator-Validator fine-tuning process, we validate table training data based on result consistency (leveraging permutation-invariance and execution-invariance). Our approach is similar in spirit to consistency-based verification methods, such as "self-consistency" and "tree-of-thoughts" (Wang et al., 2022a; Yao et al., 2024; Weng et al., 2022), but is tailored to tables.

Dual learning. Dual learning is a concept in machine learning where two related tasks are learned together via mutual reinforcement, and are used in machine translation (where the dual tasks are translating from language A to B and from B to A) (He et al., 2016; Wang et al., 2019; Sennrich and Zhang, 2019). The "duality of table tasks" we study is similar in spirit, but we develop techniques that take advantage of the unique characteristics of two-dimensional tables, which have not been previously explored.



Error Detection (Classification task T_C)

Instruction: Please examine the table below and check if any cell may be erroneous ... Please answer using JSON: {"answer": ""}.

Table:
| State | ...
| Washington | ...
| Oregon | ...
| Missisipi | ...

Completion:
The cell " Missisipi" is misspelled and should be corrected as " Mississippi". Therefore, the answer is **{"answer": "Missisipi"}**.

NL-to-SQL (Generative task T_G)

Instruction: Please examine the question below, and generate SQL code that can run on the table below ...

Question: Which team has the highest goal?

Table:
Pos	Team	Played	Win	Draw	Lost	Goals F
1	Gzira United F.C.	26	18	6	2	58
2	Gudja United F.C.	26	16	6	4	57
3	Mellieha S.C.*	26	16	6	4	51

Completion:
{SQL': 'SELECT "Team"\n FROM "table"\n ORDER BY "Goal F" DESC\n LIMIT 1'}

(a) Error-detection: Classification (b) NL-to-SQL: Generative

Figure 2: Example tasks: Error-detection, NL-to-SQL

Classification table-tasks

- **Error detection** (multi-class): check if any cell in a table column is erroneous
- **Schema matching** (binary): check if a pair of columns in two tables are related
- **Entity matching** (binary): check if two rows refer to the same entity
- **Column type annotation** (multi-class): determine the type of a column from a list
- **Table fact verification** (binary): check if a statement about a table is true or not

...

Generative table-tasks

- **NL-to-Code** (SQL, R, Pandas, ...): translate natural-language questions to executable code on a table
- **Data transformation by-example** (SQL, R, Pandas, ...): generate code for data transformation, based on given input/output examples
- **Table question answering**: answer a natural language question based on a table
- **Data imputation**: fill in missing values in a table, based on table context
- **Table summarization**: summarize a table using natural language

...

Table 1: List of table-tasks: classification and generative

3 Table-Specialist: Method

Preliminary: Generative & classification tasks. Many table tasks have been studied in the literature. Table 1 shows a list of common examples.

Some of these tasks can be "*classification*" in nature, where the output has to come from a pre-defined set of options. Examples of classification table tasks include Error detection (checking whether any cell in a table may be an error) (Chu et al., 2016; Wang and He, 2019; Heidari et al., 2019), Schema matching (checking whether two table columns match) (Rahm and Bernstein, 2001; Madhavan et al., 2001; Koutras et al., 2021), etc.

Note that table tasks may also be "*generative*" in nature, where new output needs to be generated. The examples here include NL-to-Code (generating code that can be executed on a table for a given natural language question) (Zhao et al., 2024; Lipman et al., 2024; Zhong et al., 2017), where the generated code can be in a target DSL such as SQL, R, Pandas, and Scala, etc.

Following prior work (Li et al., 2024b; Zhang et al., 2024b), to use language models to solve table tasks, we represent each instance of a table task as an "(`instruction, table, completion`)" triple:

Definition 1. [Table tasks]. An instance of a table task, denoted by t, is defined as a triplet $t = (I, R, C)$, where I is the natural language instruction to describe the task, R is the input table on which the task is performed, C is the expected completion by following the instruction I and performing the task on table R. □

We give concrete examples of table tasks below.

Example 1. [Table tasks]. Figure 2 (a) shows an instance of the Error detection task, which is a classification task that identifies values in a table column that may be erroneous. Figure 2(b) shows an example generative table task, NL-to-SQL. □

Goal: specialist models that can generalize. Recall that a key motivation of this work is the observation that fine-tuning on the training-split of a narrow benchmark dataset D (often manually labeled on a small scale) usually leads to over-fitting even on large language models. We would like to build "specialist models" that specialize in a given type of table task T (say NL-to-Code or Error detection), that crucially *generalize to new and unseen dataset of the task T.*

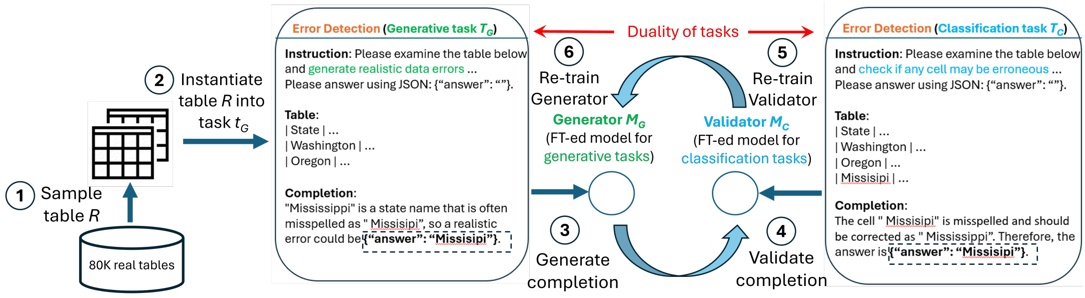
Given the promise of language models in generating synthetic data, to train small and specialized language models (e.g., code, and embedding models) (Wang et al., 2023; Li et al., 2023; Adler et al., 2024), we explore a similar direction to train table-specialist models using large amounts of synthetically generated training data (beyond the scale possible with manual labeling), for generalizability.

Challenge: validate training data. An obvious challenge is that vanilla language models, denoted by M, do not always generate high-quality training data for task T, because the task may be unfamiliar to language models, or the DSL (e.g., R or Scala in NL-2-R and NL-2-Scala) may be less familiar, etc.

Training data directly generated by vanilla language models are often far from perfect, which call for ways to "validate" synthetic training data automatically generated by language models, before it can be reliably used to fine-tune specialist models.

Our approach: validate training data using "task duality". To systematically validate training data, we observe that there is a natural "*duality*" in table tasks. Specifically, for each classification table task T_C, we can construct a "dual" generative task T_G, and vice versa, defined as follows:

Definition 2. [Task duality]. Let T_G be a generative table task and $T_G(R)$ be an instance of the task T_G instantiated with a table R. Similarly, let T_C be a classification table task, and $T_C(f(R))$ be an instance of the task T_C, instantiated with table $f(R)$, where f is a deterministic transformation function applied to R. Let M be an oracle model that produces ground truth completions for



1 Sample table R
80K real tables
2 Instantiate table R into task t_G
Duality of tasks
6 Re-train Generator
5 Re-train Validator
Generator M_G (FT-ed model for generative tasks)
Validator M_C (FT-ed model for classification tasks)
3 Generate completion
4 Validate completion

Error Detection (Generative task T_G)

Instruction: Please examine the table below and generate realistic data errors ... Please answer using JSON: {"answer": ""}.

Table:
| State | ...
| Washington | ...
| Oregon | ...

Completion:
"Mississippi" is a state name that is often misspelled as " Missisipi", so a realistic error could be **{"answer": "Missisipi"}**.

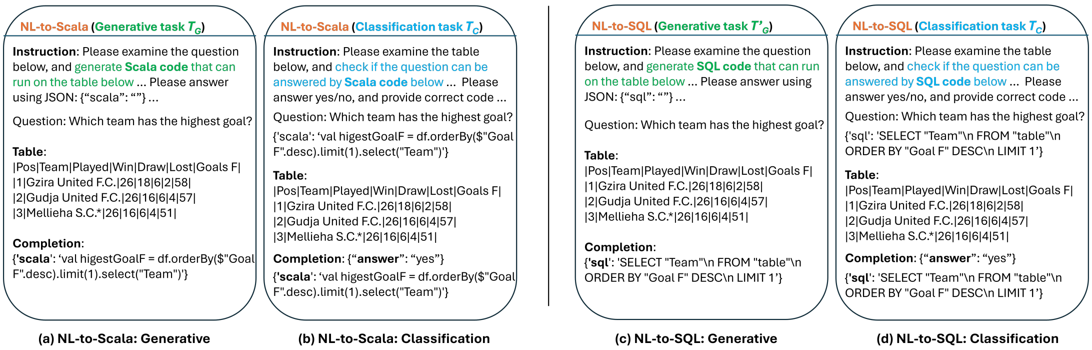
Error Detection (Classification task T_C)

Instruction: Please examine the table below and check if any cell may be erroneous ... Please answer using JSON: {"answer": ""}.

Table:
| State | ...
| Washington | ...
| Oregon | ...
| Missisipi | ...

Completion:
The cell " Missisipi" is misspelled and should be corrected as " Mississippi". Therefore, the answer is **{"answer": "Missisipi"}**.

Figure 3: Architecture of TABLE-SPECIALIST using "Generator-Validator" fine-tuning for a given task type T (Error detection in this example). (1) A real table R is sampled from a corpus of diverse tables; (2) Table R is used to instantiate an instance of the generative table task $T_G(R)$ (left box); (3) A "Generator model" M_G (initially a vanilla language-model) is used to generate completion for $T_G(R)$, in this case a possible typo error "Missisipi"; (4) The completion "Missisipi" is inserted into R, and used to instantiate a classification-version of the Error detection task T_C (right box), which is validated by a "Validator model" for the classification task M_C (initially also a vanilla language-model). If M_C consistently produces "Missisipi" for T_C, then "Missisipi" is considered validated (i.e., likely a real error); (5-6) Validated training data is then used to re-train the Generator M_G and Validator M_C, for more effective Generator and Validator models. We iteratively fine-tune M_G and M_C, by repeating steps (1)-(6) .



NL-to-Scala (Generative task T_G)

Instruction: Please examine the question below, and generate **Scala code** that can run on the table below ... Please answer using JSON: {"scala": ""} ...

Question: Which team has the highest goal?

Table:
Pos	Team	Played	Win	Draw	Lost	Goals F
1	Gzira United F.C.	26	18	6	2	58
2	Gudja United F.C.	26	16	6	4	57
3	Mellieha S.C.*	26	16	6	4	51

Completion:
{'**scala**': 'val higestGoalF = df.orderBy($"Goal F".desc).limit(1).select("Team")'}

(a) NL-to-Scala: Generative

NL-to-Scala (Classification task T_C)

Instruction: Please examine the table below, and check if the question can be answered by **Scala code** below ... Please answer yes/no, and provide correct code ...

Question: Which team has the highest goal?

{'scala': 'val higestGoalF = df.orderBy($"Goal F".desc).limit(1).select("Team")'}

Table:
Pos	Team	Played	Win	Draw	Lost	Goals F
1	Gzira United F.C.	26	18	6	2	58
2	Gudja United F.C.	26	16	6	4	57
3	Mellieha S.C.*	26	16	6	4	51

Completion: {"**answer**": "yes"}
{'**scala**': 'val higestGoalF = df.orderBy($"Goal F".desc).limit(1).select("Team")'}

(b) NL-to-Scala: Classification

NL-to-SQL (Generative task T'_G)

Instruction: Please examine the question below, and generate **SQL code** that can run on the table below ... Please answer using JSON: {"sql": ""} ...

Question: Which team has the highest goal?

Table:
Pos	Team	Played	Win	Draw	Lost	Goals F
1	Gzira United F.C.	26	18	6	2	58
2	Gudja United F.C.	26	16	6	4	57
3	Mellieha S.C.*	26	16	6	4	51

Completion:
{'**sql**': 'SELECT "Team"\n FROM "table"\n ORDER BY "Goal F" DESC\n LIMIT 1'}

(c) NL-to-SQL: Generative

NL-to-SQL (Classification task T_C)

Instruction: Please examine the table below, and check if the question can be answered by **SQL code** below ... Please answer yes/no, and provide correct code ...

Question: Which team has the highest goal?

{'sql': 'SELECT "Team"\n FROM "table"\n ORDER BY "Goal F" DESC\n LIMIT 1'}

Table:
Pos	Team	Played	Win	Draw	Lost	Goals F
1	Gzira United F.C.	26	18	6	2	58
2	Gudja United F.C.	26	16	6	4	57
3	Mellieha S.C.*	26	16	6	4	51

Completion: {"**answer**": "yes"}
{'**sql**': 'SELECT "Team"\n FROM "table"\n ORDER BY "Goal F" DESC\n LIMIT 1'}

(d) NL-to-SQL: Classification

Figure 4: Two example NL-to-Code tasks that translate natural-language questions to code: (a, b) dual versions of the NL-to-Scala task; (c, d) dual versions of the NL-to-SQL task. "Execution-invariance": observe that in (a) NL-to-Scala, and (c) NL-to-SQL, given the same question, the generated Scala and SQL code should generate identical results when executed on the same input table.

any task. The generative task T_G is said to be a *dual task* of T_C, if for any table R, we always have $M(T_G(R)) \equiv M(T_C(f(R)))$, using some fixed transformation f.[2] □

Intuitively, a task T_G is the dual of T_C, if for any table R, $M(T_G(R))$ and $M(T_C(f(R)))$ are expected to produce the same output. We illustrate duality and its construction f in more detail in the following.

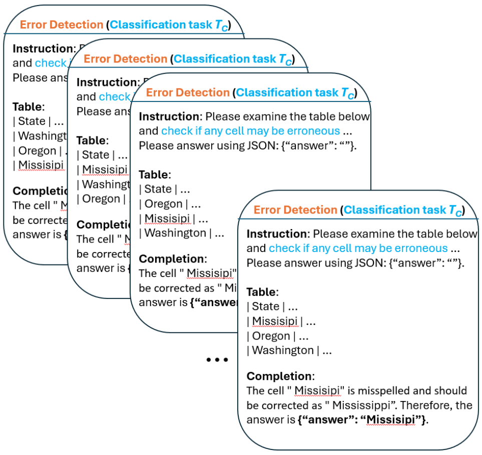
Example 2. [Task duality]. Error detection is a multi-class classification task T_C, to predict if any value in a given table column is an error, as shown on the right of Figure 3. We can construct its generative dual, T_G, shown on the left, which simply asks a model to "generate" a realist error in a table column R.

To see why T_C and T_G are dual tasks, let $T_G(R)$ be an instance of the generative Error detection task instantiated using a table R, like shown in the left-box of Figure 3. Let $c = M(T_G(R))$ be its completion, in this example c =“`Missisipi`”, a realistic typo error. Let $f(R) = insert(c, R)$ be a transformation that inserts c into R (creating the column on the right that contains the typo “`Missisipi`”). Now for task $T_C(f(R))$ (identifying errors in $f(R)$), we expect the same c =“`Missisipi`” to always be returned by an oracle model M, ensuring $M(T_G(R)) \equiv M(T_C(f(R)))$, or the two tasks always produce the same output, as shown in the dashed boxes, making the two tasks dual. □

Generator-Validator fine-tuning. Given that two dual tasks are expected to always produce the same output for the same table R (Definition 2), we leverage this duality to automatically "generate-then-validate" training data for fine-tuning.

We give an overview of our "*Generator-Validator*" fine-tuning, illustrated in Figure 3.

[2] Duality in the other direction can be defined similarly.



Error Detection (Classification task T_C)

Instruction: Please examine the table below and check if any cell may be erroneous ... Please answer using JSON: {“answer”: “”}.

Table:
| State | ...
| Missisipi | ...
| Oregon | ...
| Washington | ...

Completion:
The cell " Missisipi" is misspelled and should be corrected as " Mississippi”. Therefore, the answer is **{“answer”: “Missisipi”}**.

Figure 5: Validation by permutation invariance: we permute rows/columns (note that rows are ordered differently in the examples), and repeatedly invoke M_C to check whether a consistent completion (the typo “Missisipi”) is produced across all permutations.

Given a target classification task T_C that we want to fine-tune[3], we first construct its dual generative task T_G (and vice versa), shown as two boxes in the figure. We then iteratively fine-tune: (1) a “*Generator model*”, M_G, for the generative table task T_G, and (2) a “*Validator model*”, M_C, for the classification table-task T_C in the middle.

In Figure 3, in each iteration, we would first ① sample a real table R from a large corpus to ② instantiate a task $T_G(R)$, and then ③ invoke M_G (initially a vanilla language model) to generate a completion $c = M_G(T_G(R))$, which we know is also the expected completion for the corresponding classification task T_C, given the task-duality, which can then be used to “train” the classification model M_C. However, since such training data are not always correct, we ④ invoke M_C (initially also a vanilla language model) to systematically “validate” training data. The resulting validated training data that can then be used to ⑤ fine-tune M_C for T_C, and ⑥ fine-tune M_G for T_G, to create increasingly more capable “specialist models”, M_C and M_G, than vanilla language models.

The validation step in ④ is key to the iterative fine-tuning, where we leverage table-specific properties, such as “*permutation-invariance*” and “*execution-invariance*”, which we explain below.

Proposition 1. [Permutation-invariance]. Given a task T on a table R, let R' be any permuted version of R, whose rows and columns may be reordered. *Permutation-invariance* states that because the permuted R' does not change the semantics of the original table R, we should always have $T(R) \equiv T(R')$.[4] □

We use an example task to illustrate how the property is used in our iterative fine-tuning.

Example 3. [Permutation invariance]. We revisit Figure 3 to explain permutation-invariance in Error detection. First, both the Generator and Validator models, M_G and M_C, are initialized as a vanilla language model M. In each fine-tuning iteration, we sample a batch of k real tables (e.g., 3000), where each real table R can instantiate a generative task $t_G = T_G(R)$, by adding table R (e.g., a table with “states”) into task template T_G, as shown on the left of Figure 3.

Invoking M_G on each t_G creates an actual completion, $c = M_G(t_G)$, shown in the lower half of the box (in this case, a realistic typo error “Missisipi” that may occur in R). This completion c is then used to construct a classification-version of Error detection t_C, where we perform the transformation f (Definition 2) by inserting the generated error “Missisipi” into the original column R, to create the input table for t_C shown on the right.

Using “permutation invariance” (Property 1), we then perform repeated permutation of the table in each task t_C, creating many variants t'_C shown in Figure 5 (note that the rows inside each task are ordered differently). We then invoke M_C on each t'_C, and we expect the completion c (“Missisipi”) to be consistently produced if c is an actual error[5].

If a pair (t_C, c) can be consistently validated using M_C with permutation, the corresponding (t_G, c) and (t_C, c) pass our validation and are added to the respective training sets of M_G and M_C for this training iteration (we sample and validate using up to k real tables in an iteration). We then fine-tune M_G and M_C on the validated training data, and the iterative fine-tuning is repeated for a few iterations (up to 3), with the resulting models returned as our specialist models. □

Note that as we sample diverse real tables R to construct t_C and t_G for training (instead of using a small labeled dataset), the resulting models are less

[3] Our fine-tuning process for generative table-tasks follows the same process, thanks to the symmetry due to duality.

[4] With the exception of tasks that specifically depend on row and column orders, such as “removing the second row”, which however are uncommon (e.g., not seen in Table 1).

[5] This assumes no additional error is present in the original table – if the original table has other error, then the completion of t'_C would not be consistently c (“Missisipi”), and we will also not validate this (t_C, c) pair for downstream training.

likely to "over-fit", and can probably generalize well.

For a subset of generative table tasks (the lower half of Table 1), such as code-generation (e.g., NL-to-Code and Data-transformations), in addition to using the model-based validation (④ in Figure 3), we can also leverage a unique property of code execution on tables for validation, which we call "*execution-invariance*" described below.

Proposition 2. [Execution invariance]. Given a task T specified on a table R, let c^L be the generated code in a language L that can be executed on R to correctly solve T, and $c^{L'}$ be the generated code in a different language L' that can also solve T. Let $R_S \subseteq R$ be a table with a subset of rows of R, then for any R_S, we have $c^L(R_S) \equiv c^{L'}(R_S)$, which means that the execution of c^L and $c^{L'}$ on any $R_S \subseteq R$ should always produce identical results. □

We use NL-to-Code as an example, with Figure 4 showing two generative NL-to-Code tasks on tables, NL-to-Scala and NL-to-SQL, and their respective classification duals.

Example 4. [Execution invariance]. Figure 4(a) and (c) show two generative NL-to-Code tasks, NL-to-Scala and NL-to-SQL, respectively. Given the same question (e.g., "`which team has the highest goal`"), the generated Scala and SQL code shown at the bottom of the boxes, should always produce the same results when executed on the same table R (or its subset R_S), shown in the figure. □

Execution-invariance applies to other generative tasks involving code, such as Data-transformation, as we will see in the experiments.

Additional details. Details of our fine-tuning, such as pseudo-code, data generation strategy (e.g., using curriculum-based "textbook-like generation" (Abdin et al., 2024; Li et al., 2023)), and a discussion on "things that did not work", can be found in Appendix A.

4 Experiments

We perform extensive experiments, using GPT-3.5, GPT-4, and Llama-3.1-8B as base models. Our code is available at microsoft/Table-Specialist.

4.1 Experiment Setup

Table tasks and benchmarks. For a comprehensive evaluation, we use three sets of three generative tasks, NL-to-Code (generating SQL, R, Scala), Data-transformation (generating SQL, R, Pandas) and Table-QA; as well two classification tasks, Error detection and Schema matching, for a total of 9 table tasks. Each task is extensively evaluated using 2-5 benchmarks from the literature, as shown in Table 2, for a total of 29 evaluated benchmarks (each benchmark corresponds to a row in our main result in Table 3 and Table 4).

Table-task group	Evaluation metric	Task category	Dataset	Size
NL-to-Code (NL-to-SQL, NL-to-R, NL-to-Scala)	Execution Accuracy	easy	WikiSQL	1000
			Spider	1198
			BIRD	356
			WikiTQ	1000
			Text2Analysis	271
Table-QA	Accuracy	easy	FinQA	1000
			TableBench	424
			WikiTQ	1000
Data transformation (generating SQL, R, Pandas)	Execution Accuracy	hard	TDE	570
			Transform-text	335
Schema matching	F1	easy	DeepM	42
			WikiData	24
			HXD	468
Error detection	F1	hard	Spreadsheet-Tables	1126
			Relational-Tables	1081

Table 2: Table task and benchmark data for evaluation

Methods Compared. We compare the following:

- **Vanilla base models**: GPT-3.5[6], GPT-4[7], and Llama-3.1-8b.
- **Specialist Fine-Tuning**: Table-Specialist : Our proposed method, fine-tuned on GPT-3.5, GPT-4, and Llama-3.1-8b, respectively; FT-no-validation: Table-Specialist fine-tuned model without the validation step, to isolate its impact.
- **Generalist Fine-Tuning**: Table-GPT (Li et al., 2024b) [8]; TableLlama (Zhang et al., 2024b) [9].

We use Lora fine-tuning (Hu et al., 2021), with learning-rate multiplier of 0.5, and a batch size that is 1% of training-data-size (to ensure that each epoch has 100 steps), which is consistent across all methods.

4.2 Main Results

Fine-tuning on GPT-3.5, GPT-4 and Llama-3.1-8B. Table 3 shows detailed comparisons on all table tasks and benchmarks between each vanilla base model, and its corresponding Table-Specialist model.

It can be seen that, for all generative and classification table tasks, Table-Specialist improves

[6] We use `GPT-3.5-turbo-1106`

[7] We use `GPT-4-0613`

[8] Retrained using (Li et al., 2024c)

[9] Retrained using (Zhang et al., 2024a)

Task Type	Task	Dataset	Specialist Fine-tuning (GPT-3.5)		Specialist Fine-tuning (GPT-4)		Specialist Fine-tuning (Llama3.1-8B)	
			Vanilla	Table Specialist	Vanilla	Table Specialist	Vanilla	Table Specialist
Classification	Schema Matching	DeepM	0.984	**1**	1	1	0.857	**1**
		WikiData	0.913	**0.918**	0.952	0.952	**0.912**	0.766
		HXD	0.878	**0.897**	0.924	**0.935**	0.749	**0.852**
		Average	0.925	**0.938**	0.959	**0.965**	0.839	**0.873**
	Error Detection	Spreadsheet-Tables	0.136	**0.207**	0.403	**0.458**	0.071	**0.136**
		Relational-Tables	0.340	**0.457**	0.465	**0.529**	0.108	**0.161**
		Average	0.238	**0.332**	0.434	**0.494**	0.090	**0.148**
Generative	NL-to-SQL	WikiSQL	0.823	**0.855**	0.869	**0.874**	0.525	**0.816**
		WikiTQ	0.421	**0.513**	0.559	**0.597**	0.300	**0.449**
		Text2Analysis	0.498	**0.517**	**0.581**	0.572	0.273	**0.465**
		Spider	0.650	**0.684**	0.694	**0.704**	0.670	**0.690**
		BIRD	0.452	**0.514**	0.528	**0.556**	0.388	**0.438**
		Average	0.569	**0.616**	0.647	**0.661**	0.431	**0.572**
	NL-to-R	WikiSQL	0.567	**0.776** *	0.759	**0.827**	0.331	**0.409**
		WikiTQ	0.209	**0.404**	0.416	**0.550**	0.138	**0.257**
		Text2Analysis	0.227	**0.358**	0.382	**0.446**	0.103	**0.199**
		Spider	0.530	**0.565** *	0.563	**0.605**	0.503	**0.536**
		BIRD	0.317	**0.404**	0.430	**0.475**	0.225	**0.261**
		Average	0.370	**0.502**	0.510	**0.582**	0.260	**0.333**
	NL-to-Scala	WikiSQL	0.510	**0.794** *	0.745	**0.815**	0.359	**0.728**
		WikiTQ	0.109	**0.426** *	0.198	**0.476**	0.043	**0.188**
		Text2Analysis	0.236	**0.373** *	0.258	**0.373**	0.129	**0.214**
		Spider	0.308	**0.504** *	0.294	**0.466**	0.249	**0.356**
		BIRD	0.188	**0.360** *	0.189	**0.407**	0.100	**0.228**
		Average	0.270	**0.491** *	0.337	**0.507**	0.176	**0.343**
	Data-transformation (Pandas)	TDE	0.293	**0.346**	0.418	**0.456**	0.137	**0.161**
		Transform-Text	0.227	**0.230**	**0.297**	0.296	0.090	**0.122**
		Average	0.260	**0.300**	0.357	**0.376**	0.113	**0.142**
	Data-transformation (R)	TDE	0.200	**0.235**	0.305	**0.318**	0.063	**0.105**
		Transform-Text	0.164	**0.215**	**0.222**	0.218	0.051	**0.075**
		Average	0.182	**0.225**	0.264	**0.268**	0.057	**0.090**
	Data-transformation (SQL)	TDE	0.144	**0.168**	0.194	**0.202**	0.051	**0.089**
		Transform-Text	0.128	**0.172**	0.216	**0.227**	0.063	**0.066**
		Average	0.136	**0.170**	0.205	**0.214**	0.057	**0.078**

Table 3: Quality comparisons, between Vanilla models (GPT-3.5, GPT-4, Llama3.1-8B), and fine-tuned models. We use **bold** to indicate better performance after Table-Specialist fine-tuning, and we use * to indicate fine-tuned GPT-3.5 models that can outperform vanilla GPT-4.

Dataset	Specialist Fine-tuning (GPT-3.5)		Specialist Fine-tuning (Llama3.1-8B)	
	Vanilla	Table Specialist	Vanilla	Table Specialist
FinQA	0.222	**0.261**	0.066	**0.145**
TableBench	0.322	**0.336**	**0.277**	0.261
WikiTQ	0.546	**0.579**	0.465	**0.486**
Average	0.364	**0.392**	0.270	**0.296**

Table 4: Quality comparisons between vanilla models, and fine-tuned Table-Specialist models, on more open-ended generative task (Table-QA).

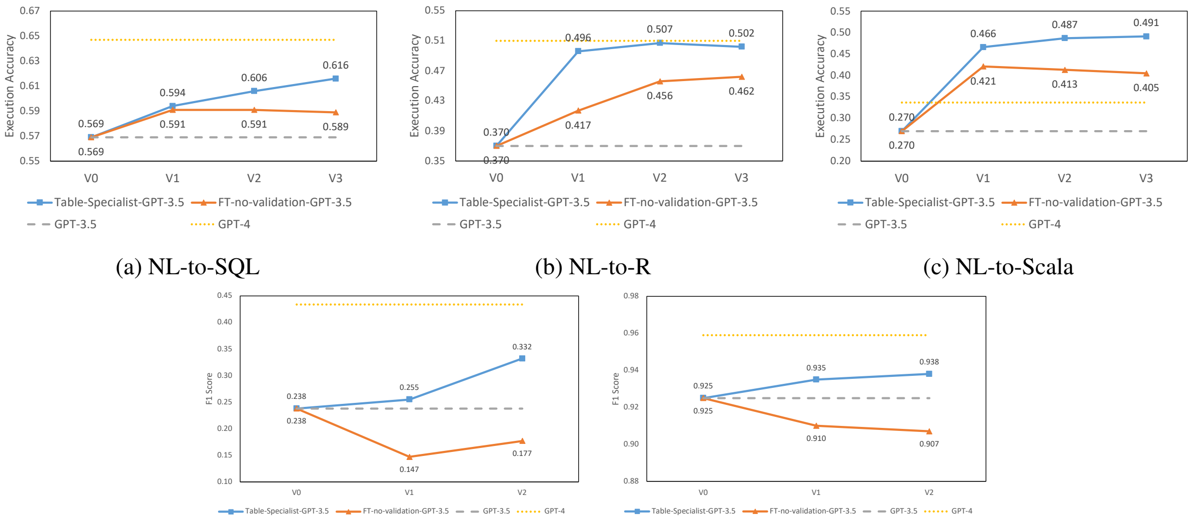
Task	Table Specialist	Generalist Fine-tuning	
		TableLlama	TableGPT
Schema Matching	**0.938**	0.918	0.896
Error Detection	**0.332**	0.227	0.222
NL-to-SQL	**0.616**	0.576	0.570
NL-to-R	**0.502**	0.373	0.370
NL-to-Scala	**0.491**	0.279	0.304
Data-transformation (Pandas)	**0.300**	0.241	0.253
Data-transformation (R)	**0.225**	0.191	0.158
Data-transformation (SQL)	**0.170**	0.146	0.137

Table 5: Comparisons between Table-Specialist and Table-Generalists (fine-tuned on GPT-3.5)

its base models (e.g., Table-Specialist-GPT-3.5 improves over GPT-3.5 on all benchmarks, and even surpassing vanilla GPT-4 on 7 benchmarks). Importantly, since we do not use the training split of any benchmark data during fine-tuning, it demonstrates that the fine-tuned models are capable of generalizing to multiple unseen benchmarks, as discussed in the introduction.

Additionally, Table 4 shows that even on more open-ended generative tasks such as Table-QA, by pairing it with related generative tasks such as NL-to-code, Table-Specialist can still operate and improve the seemingly open-ended tasks.

Table-Specialist vs. Table-Generalist. Table 5 illustrates a comparison on all table tasks between Table-Specialist and Generalist Fine-Tuning



(a) NL-to-SQL

(b) NL-to-R

(c) NL-to-Scala

(d) Error Detection

(e) Schema Matching

Figure 6: Quality of TABLE-SPECIALIST by Iteration

models (shown at the task level, in the interest of space). TABLE-SPECIALIST substantially outperforms both generalist models (TableLlama and TableGPT) on all tasks, showing the benefit of specialization over generalist models.

Iterative fine-tuning. Figure 6 shows an analysis of TABLE-SPECIALIST by fine-tuning iterations. The x-axis here represents fine-tuned iterations, where "V1", "V2" represent models from the first and second iteration, etc., and "V0" represents the base model. As we can see, TABLE-SPECIALIST demonstrates a consistent quality improvement in consecutive iterations (trending upward as we move to the right), matching or surpassing GPT-4-level quality in some cases. In comparison, FT-no-validation is substantially less effective.

Ablation	NL-to-Scala	NL-to-R	NL-to-SQL
TABLE-SPECIALIST-GPT-3.5	0.466	0.496	0.594
No-validation	0.421	0.417	0.591
No-execution-validation	0.457	0.489	0.595
No-Permutation	0.410	0.495	0.598
(Vanilla GPT-3.5)	0.270	0.370	0.569

Table 6: Ablation: (1) No validation, (2) No row/column permutation in validation, (3) No execution-based validation (use language-models as validator only).

Ablation Studies. Table 6 shows the impact of different validation strategies. We can see that not using validation at all creates clear drops in quality. Not using execution-invariance (Property 2) and permutation-invariance (Property 1) also leads to noticeable degradations, especially on low-resource tasks (e.g., Scala and R).

Additional experiments. We present additional experimental results, such as cost analysis and sensitivity analysis, in Appendix B.

5 Conclusions and Future Work

In this work, we develop a new fine-tuning approach TABLE-SPECIALIST specifically designed for table tasks. We show that it can fine-tune small models specialized for individual table tasks while still being performant and generalizable. Future directions include testing the method on additional base models and tasks beyond table tasks.

Limitations

Tasks not best suited for TABLE-SPECIALIST. We want to point out that not all table-tasks are well suited for the proposed approach. For example, this approach is not directly applicable to tasks that do not have precise "ground-truth", such as table summarization (Hancock et al., 2019; Zhang et al., 2020; Chen et al., 2013), as the lack of ground-truth makes it hard to perform validation easily.

There are also tasks that naturally come with ample training data, for which our approach would not be needed. For example, the task of Data-imputation (Mayfield et al., 2010; Biessmann et al., 2019) predicts the value for a missing cell in a table, where training data can be easily obtained (by masking out random cells in real tables, and use their ground-truth values for training). For such tasks, it would not be necessary to use Generator-Validator for fine-tuning.

Nevertheless, Generator-Validator fine-tuning in TABLE-SPECIALIST applies to a broad range of tasks with precise ground-truth that traditionally require careful manual-labeling, such as Error detection, Schema matching, NL-to-Code, and Data-transformations, as well as more open-ended tasks such as Table-QA, like we show in our experiments (Table 3 and Table 4).

References

Deepmatcher datasets. https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md.

Log probabilities for GPT models. https://platform.openai.com/docs/api-reference/chat/create#chat-create-logprobs.

OpenAI: ChatGPT. https://openai.com/blog/chatgpt.

OpenAI pricing. https://openai.com/api/pricing/.

OpenAI Pricing. https://openai.com/api/pricing/. Accessed: 2024-07-02.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, and 1 others. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Bo Adler, Niket Agarwal, Ashwath Aithal, Dong H Anh, Pallab Bhattacharya, Annika Brundyn, Jared Casper, Bryan Catanzaro, Sharon Clay, Jonathan Cohen, and 1 others. 2024. Nemotron-4 340b technical report. *arXiv preprint arXiv:2406.11704*.

Felix Biessmann, Tammo Rukat, Philipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. 2019. Datawig: Missing value imputation for tables. *J. Mach. Learn. Res.*, 20(175):1–6.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Jieying Chen, Jia-Yu Pan, Christos Faloutsos, and Spiros Papadimitriou. 2013. Tsum: fast, principled table summarization. In *Proceedings of the Seventh International Workshop on Data Mining for Online Advertising*, pages 1–9.

Qixu Chen, Yeye He, Raymond Chi-Wing Wong, Weiwei Cui, Song Ge, Haidong Zhang, Dongmei Zhang, and Surajit Chaudhuri. 2025. Auto-test: Learning semantic-domain constraints for unsupervised error detection in tables. *Proceedings of the ACM on Management of Data*, 3(3):1–27.

Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*, pages 2201–2206.

Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, Pradap Konda, Yash Govind, and Derek Paulsen. The magellan data repository. https://sites.google.com/site/anhaidgroup/useful-stuff/the-magellan-data-repository?authuser=0.

Haoyu Dong, Zhoujun Cheng, Xinyi He, Mengyu Zhou, Anda Zhou, Fan Zhou, Ao Liu, Shi Han, and Dongmei Zhang. 2022. Table pre-training: A survey on model architectures, pre-training objectives, and downstream tasks. *arXiv preprint arXiv:2201.09745*.

Haoyu Dong and Zhiruo Wang. 2024. Large language models for tabular data: Progresses and future directions. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2997–3000.

Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How large language models will disrupt data management. *Proceedings of the VLDB Endowment*, 16(11):3302–3309.

Braden Hancock, Hongrae Lee, and Cong Yu. 2019. Generating titles for web tables. In *The World Wide Web Conference*, pages 638–647.

William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *ACM SIGPLAN Notices*, 46(6):317–328.

Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. *Advances in neural information processing systems*, 29.

Junxian He, Jiatao Gu, Jiajun Shen, and Marc'Aurelio Ranzato. 2019. Revisiting self-training for neural sequence generation. *arXiv preprint arXiv:1909.13788*.

Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (tde) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177.

Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*, pages 829–846.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César Hidalgo. 2019. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1500–1508.

Keti Korini and Christian Bizer. 2023. Column type annotation using chatgpt. *arXiv preprint arXiv:2306.00745*.

Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024a. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024b. Table-gpt: Table fine-tuned GPT for diverse table tasks. *Proc. ACM Manag. Data*, 2(3):176.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024c. Table-GPT Training Data. https://huggingface.co/datasets/LipengCS/Table-GPT.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

Tavor Lipman, Tova Milo, Amit Somech, Tomer Wolfson, and Oz Zafar. 2024. Linx: A language driven generative system for goal-oriented automated data exploration. *arXiv preprint arXiv:2406.05107*.

Mark Lutz. 2001. *Programming python*. " O'Reilly Media, Inc.".

Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. 2001. Generic schema matching with cupid. In *vldb*, volume 1, pages 49–58.

Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. 2010. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 75–86.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707*.

Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

Erhard Rahm and Philip A Bernstein. 2001. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10:334–350.

Erhard Rahm, Hong Hai Do, and 1 others. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13.

Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database management systems*. McGraw-Hill, Inc.

Rico Sennrich and Biao Zhang. 2019. Revisiting low-resource neural machine translation: A case study. *arXiv preprint arXiv:1905.11901*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Yuan Sui, Jiaru Zou, Mengyu Zhou, Xinyi He, Lun Du, Shi Han, and Dongmei Zhang. 2023. Tap4llm: Table provider on sampling, augmenting, and packing semi-structured data for large language model reasoning. *arXiv preprint arXiv:2312.09039*.

Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher D Manning. 2023. Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback. *arXiv preprint arXiv:2305.14975*.

Yuzhang Tian, Jianbo Zhao, Haoyu Dong, Junyu Xiong, Shiyu Xia, Mengyu Zhou, Yun Lin, José Cambronero, Yeye He, Shi Han, and 1 others. 2024. Spreadsheetllm: Encoding spreadsheets for large language models. *arXiv preprint arXiv:2407.09025*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Pei Wang and Yeye He. 2019. Uni-detect: A unified approach to automated error detection in tables. In *Proceedings of the 2019 International Conference on Management of Data*, pages 811–828.

Renxi Wang, Haonan Li, Xudong Han, Yixuan Zhang, and Timothy Baldwin. 2024. Learning from failure: Integrating negative examples when fine-tuning large language models as agents. *arXiv preprint arXiv:2402.11651*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022a. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yiren Wang, Yingce Xia, Tianyu He, Fei Tian, Tao Qin, ChengXiang Zhai, and Tie-Yan Liu. 2019. Multi-agent dual learning. In *Proceedings of the International Conference on Learning Representations (ICLR) 2019*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022b. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. 2022. Large language models are better reasoners with self-verification. *arXiv preprint arXiv:2212.09561*.

Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. 2020. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10687–10698.

Junjie Xing, Yeye He, Mengyu Zhou, Haoyu Dong, Shi Han, Lingjiao Chen, Dongmei Zhang, Surajit Chaudhuri, and HV Jagadish. 2025. Mmtu: A massive multi-task table understanding and reasoning benchmark. *arXiv preprint arXiv:2506.05587*.

Junjie Xing, Yeye He, Mengyu Zhou, Haoyu Dong, Shi Han, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-llm-specialist: Language model specialists for tables using iterative generator-validator fine-tuning. *Preprint*, arXiv:2410.12164.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium. Association for Computational Linguistics.

Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. 2018. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.

Shuo Zhang, Zhuyun Dai, Krisztian Balog, and Jamie Callan. 2020. Summarizing and exploring tabular data in conversational search. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1537–1540.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024a. TableInstruct. https://huggingface.co/datasets/osunlp/TableInstruct/.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024b. Tablellama: Towards open large generalist models for tables. *Preprint*, arXiv:2311.09206.

Yu Zhang, Mei Di, Haozheng Luo, Chenwei Xu, and Richard Tzong-Han Tsai. 2024c. Smutf: Schema matching using generative tags and hybrid features. *arXiv preprint arXiv:2402.01685*.

Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. Nl2formula: Generating spreadsheet formulas from natural language queries. *arXiv preprint arXiv:2402.14853*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

Xiaojin Zhu and Andrew B Goldberg. 2022. *Introduction to semi-supervised learning*. Springer Nature.

A Details on Generator-Validator Finetuning

A.1 TABLE-SPECIALIST: Classification Task

Many table tasks studied in the literature are classification in nature, which can be binary-classifications (e.g., Schema matching, Entity-matching, Table-fact-verification), or multi-class classification (e.g., Error detection, Column-type-annotation, etc.).

Given a target classification table-task T_C that we want to fine-tune, and its dual generative table-task T_G that we can construct, in this section, we describe how the Generator-Validator framework can fine-tune for T_C.

Algorithm 1: Generator-Validator fine-tuning

```
Input: A corpus of real table $\mathcal{R}$, a vanilla language-model
    $M$, a generative table-task $T_G$, a corresponding classifi-
    cation table-task $T_C$
Output: Fine-tuned specialist model $M_G$ for task $T_G$, and
    $M_C$ for task $T_C$
1:  $M_G \leftarrow M$ // initialize the generative model $M_G$ as
    vanilla $M$
2:  $M_C \leftarrow M$ // initialize the classification model $M_C$
    as vanilla $M$
3:  for $i$ in 1 to $k$ iterations do
4:      Train$_G \leftarrow \{\}$ // initialize the validated training
        set for $T_G$
5:      Train$_C \leftarrow \{\}$ // initialize the validated training
        set for $T_C$
6:      for $j$ in 1 to step-size do
7:          Sample $R \in \mathcal{R}$ // sample a real table
8:          Instantiate $t_G \leftarrow T_G(R)$ // instantiate a
            generative task $t_G$ using $R$
9:          $c \leftarrow M_G(t_G)$ // invoke $M_G$ to compute the
            completion $c$ for $t_G$
10:         Construct $t_C \leftarrow T_C(R, c)$ // construct a classification
            task $t_C$ with $R$, $c$
            // check $c$ is a valid completion of $t_G$, by calling
            Validate()
11:         if Validate($M_C$, $t_C$, $c$) then
12:             Train$_G \leftarrow$ Train$_G \cup (t_G, c)$ // add the validated
                $(t_G, c)$ into Train$_G$
13:             Train$_C \leftarrow$ Train$_C \cup (t_C, c)$ // add the validated
                $(t_C, c)$ into Train$_C$
14:     Fine-tune $M_G$ using Train$_G$ // fine-tune $M_G$ using
        validated training data
15:     Fine-tune $M_C$ using Train$_C$// fine-tune $M_C$ using
        validated training data
    return $M_G, M_C$ // return fine-tuned models $M_G, M_C$
```

Algorithm 1 shows the general steps of the Generator-Validator approach. We start by initializing both the generative model M_G for the generative task T_G, and the classification model M_C for the classification task T_C, as a vanilla language model M (Line 1-2). We then start our iterative fine-tuning (Line 3), by first initializing training set for T_G and T_C as empty sets (Line 4-5). In each fine-tuning iteration, we iteratively perform *step-size* number of sampled steps (Line 6), where each time, we sample a real table R from the corpus (Line 7), which we use to instantiate an instance of the generative task $t_G = T_G(R)$ (Line 8). We then invoke M_G on t_G, to produce a completion c (Line 9). We use c and R to construct a corresponding classification task t_C (Line 10). At this point, we perform the crucial validation step by calling the Validate() subroutine (Line 11, which calls Algorithm 2 and will be explained next). Once the validation passes, we add (t_G, c) and (t_C, c) as validated training examples for T_G and T_C, respectively, because by duality c will be a correct completion for both t_G and t_C (Line 12-13). After performing step-size number of samples, the validated training data will be used to fine-tune M_G and M_C (Line 14-15), to conclude one iteration of the fine-tuning process. We repeat k such iterations, and return the resulting M_G and M_C as our TABLE-SPECIALIST models.

Algorithm 2: Validate(M_C, t_C, c): validate for classification tasks

```
Input: A classification model $M_C$, an instance of classifica-
    tion task $t_C$, and its expected output $c$
Output: True or False // validate whether $c$ is the correct
    completion for $t_C$
1:  $R \leftarrow t_C.R$ // get the table $R$ used in task $t_C$
2:  for $i$ in 1 to $N$ do
3:      $t'_C \leftarrow T_C(R')$ // instantiate a new $T_C$ task, using
        the permuted $R'$
4:      $c' \leftarrow M_C(t'_C)$ // get completion $c'$ for $t'_C$, using
        classification model $M_C$
5:      if ($c' \neq c$) then
6:          return False // Not-validated: unsure if $c$ is
            correct completion for $t_C$
    return True // Validated: $c$ is likely the correct
    completion for $t_C$
```

Algorithm 2 shows the validation subroutine (Line 11 of Algorithm 1), which is necessary for the following reason. Recall that $c = M_G(t_G)$ is a completion generated by invoking M_G on task t_G, which we expect to also be the completion of the corresponding dual task t_C (by task-duality in Definition 2), such that we can use (t_C, c) as training data to train model M_C for our target classification task T_C. However, M_G is often not perfect in many table-tasks as we discussed, so that $c = M_G(t_G)$ may not be the correct completion for t_G, and thus also not the correct completion for t_C, in which case (t_C, c) pairs should not be used for training. We therefore use the subroutine in Algorithm 2 for this validation.

A.2 TABLE-SPECIALIST: Generative Tasks

In this section, we describe how Generator-Validator fine-tuning can be applied to generative table-tasks (the lower half of Table 1), such as NL-to-Code and Data-transformation, etc. Figure 4 shows two generative NL-to-Code tasks on tables, NL-to-Scala and NL-to-SQL, and their respective classification duals.

Our fine-tuning process for generative table-tasks uses the same Generator-Validator approach in Algorithm 1, thanks to the the symmetry between generative/classification tasks in our setup.

As additional opportunities, we observe that for a subset of generative table-tasks, such as code-generation (e.g., NL-to-Code and Data-transformations), where the target code can be in languages such as SQL, R, Scala, Pandas, in addition to using the model-based validation in Line 11 of Algorithm 1 (which invokes Algorithm 2), we can also leverage a unique property of executing code on tables for validation that we call "*execution-invariance*" described in Proposition 2.

The execution-invariance property provides us with an alternative to model-based validation (Algorithm 2), by using execution-based validation, which we explain in Algorithm 3 below.

Algorithm 3: Validate($M_G^L, M_G^{L'}, t_G$): for code generative tasks

```
Input: A generative model $M_G^L$ for generating code in a target language $L$, another generative model $M_G^{L'}$ for generating code in a second language $L'$, an instance of classification task $t_G$
Output: True or False
$R \leftarrow t_G.R$ // get the table $R$ used in task $t_G$
$c^L \leftarrow M_G^L(t_G)$ // generate target code $c^L$ in language $L$
$c^{L'} \leftarrow M_G^{L'}(t_G)$ // generate target code $c^{L'}$ in language $L'$
for $i$ in 1 to $N$ do
    $R_S \leftarrow$ Sample($R$) // sample rows in table $R$
    $r \leftarrow$ Execute($c^L, R_S$) // execute $c^L$ on table $R_S$ to get $r$
    $r' \leftarrow$ Execute($c^{L'}, R_S$) // execute $c^{L'}$ on table $R_S$ to get $r'$
    if ($r \neq r'$) then
        return False // Not-validated: unsure if $c^L$ is correct completion for $t_G$
return True // Validated: $c^L$ is likely a correct completion for $t_G$
```

In Algorithm 3, we are given a generative model M_G^L that can generate code on tasks t_G in a target language L (e.g., NL-to-Scala). We use a second model $M_G^{L'}$ that generates code for the same task t_G but in a different language L' (e.g., NL-to-SQL), to validate code generated by M_G^L.

We start by assigning R as the table used in t_G, then invoke M_G^L and $M_G^{L'}$ (both are initially vanilla language models), to generate code c^L and $c^{L'}$ respectively. Then in N iterations, we repeatedly sample rows to generate $R_S \subseteq R$, and execute c^L and $c^{L'}$ on R_S, to produce results r and r', respectively. If in any iteration we have $(r \neq r')$, then by execution-invariance we know that c^L and $c^{L'}$ are not semantically equivalent, and at least one of the two is incorrect, which is why we return "`False`" to signify that c^L cannot be validated so that it will not be used in training later. Otherwise, if we cannot find contradictions in N iterations, we consider (t_G, c^L) a valid training example and return "`True`" for this data point to fine-tune M_G^L. Note that $(t_G, c^{L'})$ is also a valid training example, so that we can fine-tune $M_G^{L'}$ for a different language L' in parallel.

We illustrate Algorithm 3 using NL-to-Code as an example.

Example 5. Consider the task of NL-to-Scala, or generating Scala code that can run on Spark, as shown in Figure 4(a). Like in Figure 3, as pre-processing steps, we would first sample a real table R, and then ask language-models to brainstorm a question relevant to table R, e.g., "`which team has the highest goal`" for the table in the figure, to create a generative task t_G. The classification version of the task is shown in Figure 4(b), which asks a model to predict whether a code snippet can execute to answer a given natural-language question. With these two tasks, we can already perform Generator-Validator fine-tuning using Algorithm 1 and 2.

Leveraging execution-invariance, we can perform a different type of validation, that invokes Algorithm 3 (in place of Algorithm 2). Specifically, when validating training data (Line 11 of Algorithm 1), we invoke Algorithm 3, where we use the same task, but require code to be generated in a different language – Figure 4(c) shows an NL-to-SQL task that directly corresponds to Figure 4(a) but requires generated code to be in SQL.

Let M_G^L be the NL-to-Scala model that we iteratively fine-tune, and $M_G^{L'}$ be a NL-to-SQL model (which can be a vanilla language-model, or another model that we also iteratively fine-tune in lockstep), we can then proceed to invoke Algorithm 3. We first generate code in both Scala and

SQL for the same question, like shown in the bottom of Figure 4(a) and (c), and then execute both Scale and SQL repeatedly on sub-samples $R_S \subseteq R$, to compare their execution results. If we cannot find contradictions in any iteration, we consider (t_G, c^L) and $(t_G, c^{L'})$ validated, which we can use to iterative fine-tune M_G^L and $M_G^{L'}$. (This in effect changes the right-half of the architecture in Figure 3, by replacing the mode-based validation, into an execution-based validation). □

Note that the execution-based validation applies to other generative tasks involving code, such as Data-transformation by-example, or generating code to perform transformations specified by input/output examples, using a target language (e.g., SQL, R, Scala, etc.).

A.3 "Textbook-like" generation

There are additional details in our initial data generation process, where we use a curriculum-guided process to direct language-models to compose textbook constructs so that they can generate diverse questions of varying levels of difficulty that are relevant to a given table R that we will explain below (Abdin et al., 2024; Li et al., 2023).

Recall that for generative tasks such as NL-to-Code and Data-transformations, there is an initial preprocessing step in which we need to generate a reasonable "task" t for a sampled table $R \in \mathcal{C}$, so that the the question t can then become part of the instruction and used as training data to fine-tune language models (e.g., for NL-2-Code, this t would be a natural language question that needs to be answered based on the content of R, for Data-transformation, this t would be a ground-truth transformation that we want models to predict based on input/output examples).

While language-models can by themselves generate reasonable tasks t given a table R, we find benefit in guiding language-models towards constructing diverse t by composing basic building-blocks from programming language "textbooks".

For example, for NL-2-Code, we find it beneficial to decompose the question-generation task, into an explicit list of requirements based on atomic SQL constructs, such as where, group-by, order-by, etc., using database textbooks (Ramakrishnan and Gehrke, 2002), and then ask language-models to brain-storm a question using a given table R, based on the constraints, like below:

- *THREE (3)* of filtering predicate(s) in WHERE clause
- *TWO (2)* GROUP BY clause, with aggregation function
- *ONE (1)* ORDER BY command
- ...

Note that the numbers 3/2/1 shown above are examples, which are randomly sampled from a range of $[0, k]$, and dynamically inserted into the prompt when generating a question q on a table R. This produces diverse questions with varying degrees of difficulty, that can be answered using SQL or otherwise. (In comparison, if language models are asked to generate questions unconstrained, they tend to produce similar questions on different tables that are less diverse, which we find to be less effective as training data).

Similarly, for Data-transformation, we decompose the task of generating reasonable transformations that can be performed on a given table R, also into a list of atomic building blocks, using basic Python constructs, such as string-transformation (split, concatenate, sub-string, etc.), number transformation, array transformations, etc., using Python textbooks (Lutz, 2001). We sample requirements from the list, in order for language-models to generate diverse transformation examples that can be used as training data.

Details of the prompts used in generating task t for NL-2-Code and Data-transformation, can be found in our technical report (Xing et al., 2024).

A.4 Things we tried but were not effective

In addition to what is reported, we also tried many things that did not turn out to be effective, which we will report below.

Since it is standard to use confidence scores as soft-labels in self-supervised and semi-supervised learning (He et al., 2019; Zhu and Goldberg, 2022; Xie et al., 2020), in TABLE-SPECIALIST we also tried to extract confidence scores of training examples from language-models during the generation process. For example, we used log-probabilities (llm) as well as verbalization techniques (Sui et al., 2024; Tian et al., 2023) to extract confidence from language-models, which we use to find confident training examples in the self-training / iterative fine-tuning process, which however was not always beneficial.

During the data validation process, we produce lots of negative (invalidated) examples, in addition to positive (validated) examples. In one variant of our fine-tuning, we try to use both positive (validated) and negative (invalidated) examples (e.g., using a format suggested in (Wang et al., 2024)), to prefix positive and negative examples with leading special-tokens, such as [POS] and [NEG], respectively, which was not helpful in our experiments.

Since some of the table-tasks we test are pretty challenging (e.g., Data-transformations, which requires trial-and-test, and reflect on previous mistakes), we also tried agentic self-reflection style fine-tuning using trajectories, by allowing language-models to make multiple attempts in generating transformation-programs, each looking at the output from previous attempts to reflect on previous errors (e.g., compilation errors in previous execution, or output from a previous execution does not match the intended output), similar to (Shinn et al., 2024) in NLP tasks. While it provides modest benefit in terms of overall success rate for challenging tasks like Data-transformation, it substantially increases the latency for training data generation, and complicates the overall architecture, which we decided not to include in the end.

B Additional Experiments

B.1 Latency and Cost Comparisons

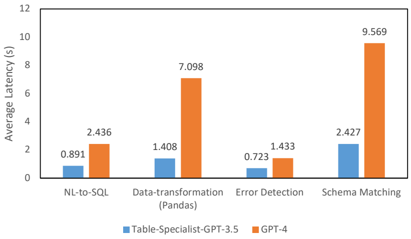
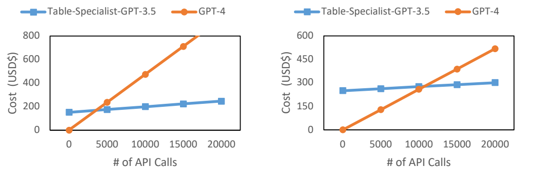


Figure 7: Average Latency for TABLE-SPECIALIST (fine-tuned on GPT-3.5) and vanilla GPT-4

In Figure 7, we compare the average latency of TABLE-SPECIALIST models (fine-tuned on GPT-3.5) and GPT-4, on various task, averaged over all benchmark test cases. Because the fine-tuned TABLE-SPECIALIST models are smaller, on average they are 3.42 times faster than vanilla GPT-4 (while still having comparable quality). Figure 1 shows another analysis for NL-to-Code tasks, with similar latency reductions. Since serving online queries and ensuring interactivity is key in many user-facing workloads, this highlights a crucial benefit of TABLE-SPECIALIST as it allows us to employ smaller models to reduce latency significantly.



(a) Schema Matching (b) Data-transformation

Figure 8: Total Cost Analysis: TABLE-SPECIALIST v.s. GPT-4

In Figure 8, we compare the cost[10] of fine-tuning and serving TABLE-SPECIALIST using GPT-3.5, vs. serving directly using vanilla GPT-4, on two table tasks (results for other tasks are similar).

The detailed unit price is listed in Table 7. We estimate the cost per API call using the average number of prompt and completion tokens for each tasks, as listed in Table 8.

Table 7: The Unit Price for Inference and Training, Per 1K Tokens, for Vanilla GPT-3.5, GPT-4, and Fine-tuned GPT-3.5. As of July 2, 2024 (ope)

Model	Input	Output	Training
GPT-3.5	0.001	0.002	0.008
FT(GPT-3.5)	0.003	0.006	N.A.
GPT-4	0.03	0.06	N.A.

Table 8: Average Number of Prompt and Completion Tokens, and Average Latency for TABLE-SPECIALIST and GPT-4

Task	Average # of Tokens		Average Latency (s)	
	Prompt	Completion	TABLE-SPECIALIST	GPT-4
NS	969	38	0.891	2.436
R2RP	678	92	1.408	7.098
ED	701	10	0.723	1.433
SM	1168	206	2.427	9.569

We can see in Figure 8 that TABLE-SPECIALIST-GPT-3.5 has to pay an upfront cost of fine-tuning, which is why it starts with a non-zero cost (on y-axis) to serve the first query (on x-axis). This however, is amortized over future queries, and takes less than 5000 queries (for Schema matching) or 10000 queries (for Data-transformation) for TABLE-SPECIALIST to break even with using vanilla GPT-4 directly. We argue that the cost saving in the long run, together with significant latency reductions, makes TABLE-SPECIALIST a viable option, especially in user-facing online settings.

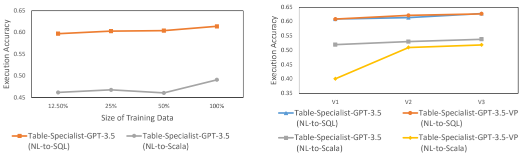
[10] We use the published pricing (pri) to calculate the cost of fine-tuning and inference.

Table 9: Ablation study: No "textbook-like" constrained task generation.

Ablation	NL-to-SQL	NL-to-R	NL-to-Scala	Transform-Pandas	Transform-R	Transform-Scala
TABLE-SPECIALIST	0.609	0.498	0.510	0.267	0.222	0.133
NoTextBook	0.601	0.497	0.501	0.257	0.200	0.135

B.2 Sensitivity Analysis

We perform various types of sensitivity analysis in TABLE-SPECIALIST.



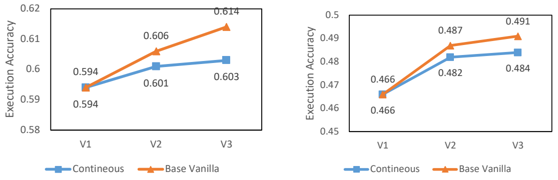
(a) Vary Training Size (b) Vary Prompt Templates

Figure 9: Sensitivity analysis

Vary the Amount of Training Data. Figure 9a shows TABLE-SPECIALIST quality, when we vary the amount of training data produced by the Generator from 100% to 50%, 25% and 12.5% of the original data (x-axis). We can see that increasing the amount of training data generally has a positive effect on result quality.

Vary Prompt Templates. To test the robustness of TABLE-SPECIALIST, we vary our prompt templates used in each task, by giving our original prompt to ChatGPT and asking it to paraphrase into five different prompts, for the NL-to-SQL and NL-to-Scala tasks. Figure 9b shows that using variants of the prompt (abbreviated as VP in the figure), lead to comparable quality.

Vary the Base Model for Fine-Tuning. We test two alternatives of iterative fine-tuning, where in each iteration, we initialize the base model either as the vanilla model (e.g., GPT-3.5), or the model from the last iteration (continuous fine-tune). Figure 10 shows that using vanilla GPT as the base models are consistently better than using the checkpoint from the last iteration (continuous fine-tune), for both NL-to-SQL and NL-to-Scala.



(a) NL-to-SQL (b) NL-to-Scala

Figure 10: Vary Base Model

No "textbook-like" Constrained Task Generation. We removed the "textbook" constrained task generation for the two generative tasks, and report resulting quality in Table 9. We observe that having textbook-like curriculum-guided data generation improves the diversity of training data, and generally has a positive effect on the final model quality.