HITSZ-HLT at SemEval-2025 Task 8: Multi-turn Interactive Code Generation for Question Answering on Tabular Data

Jun Wang[1]* **Feng Xiong**[1]* **Hongling Xu**[1] **Geng Tu**[1] **Ruifeng Xu**[1]†
[1]Harbin Institute of Technology, Shenzhen, China
`23s051031@stu.hit.edu.cn, xuruifeng@hit.edu.cn`

Abstract

This paper introduces the system developed by the HITSZ-HLT team for SemEval-2025 Task 8: DataBench, Question-Answering over Tabular Data. The primary objective of Table Question Answering (TableQA) is to provide accurate answers to user queries by interpreting and understanding tabular data. To address this, we propose the **M**ulti-turn **I**nteractive **C**ode Generati**O**n (MICO) framework. Specifically, MICO employs code generation as proxy task for TableQA and integrates feedback from the execution of the generated code via multi-turn dialogue process, thereby guiding the model towards self-correction. Experimental results demonstrate the effectiveness of our framework, achieving notable performance with a rank of 4/38 on the DataBench and 5/38 on the DataBench lite.

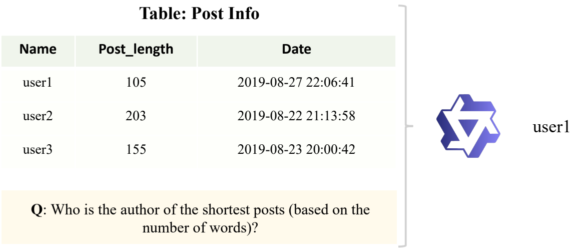


Table: Post Info

Name	Post_length	Date
user1	105	2019-08-27 22:06:41
user2	203	2019-08-22 21:13:58
user3	155	2019-08-23 20:00:42

Q: Who is the author of the shortest posts (based on the number of words)?

Figure 1: An example of TableQA system.

1 Introduction

Table Question Answering (TableQA) (Pal et al., 2023; Hu et al., 2024; Zhao et al., 2024; Osés-Grijalba et al., 2025) has gained significant attention due to the extensive use of tabular data in various domains (Jin et al., 2022; Nan et al., 2022). The primary objective of TableQA is to accurately interpret and process tabular data, enabling autonomous generation of answers to user queries. As shown in Fig. 1, the goal of the model to leverage the information provided within the table to identify the author of the shortest post. By empowering machines to reason over structured data in tables, TableQA systems seek to offer a more effective and efficient approach to interacting with large datasets (Giang et al., 2024). While this task holds great potential, it is also accompanied by several challenges (Wu et al., 2024). The inherent complexity of tabular data, which includes large datasets (Su et al., 2024), unordered structures, and high-precision numerical values, presents significant obstacles in generating accurate and efficient responses to queries.

In this paper, we propose **M**ulti-turn **I**nteractive **C**ode Generati**O**n (MICO). Specifically, we leverage code generation tasks as a surrogate for TableQA to reduce the model's complexity in processing long-context inputs and performing precise numerical computations. Initially, we integrate the table's metadata along with a few sample instances into the prompt, thereby directing the model to generate code. The generated code is then executed within a sandboxed environment[1] to acquire feedback. In subsequent rounds of interaction, if the model determines the code has been successfully executed, it will regenerate the structured output and deliver the final result. Conversely, if the execution is deemed unsuccessful, the model will engage in self-correction and restart the process.

Additionally, we conducted a thorough evaluation of our approach using the DataBench and DataBench Lite datasets, which provided strong evidence of its effectiveness. This extensive validation process resulted in impressive performance, securing a ranking of 4th out of 38 participants on the DataBench leaderboard and 5th out of 38 on the DataBench Lite leaderboard, as reported on the official rankings. Moreover, additional experiments further confirm the effectiveness of each component.

---

* Equal contribution.

†Corresponding author.

[1] https://github.com/vndee/llm-sandbox

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 2318–2324
July 31 - August 1, 2025 ©2025 Association for Computational Linguistics

2 Related Work

Existing TableQA research primarily focuses on two main directions: semantic parsing and query generation. Research in the semantic parsing direction involves joint training of natural language questions and tabular text data, followed by fine-tuning for specific tabular tasks, enabling the model to understand the semantic information in the table and provide accurate answers (Mueller et al., 2019; Eisenschlos et al., 2020; Zhou et al., 2022; Hu et al., 2024). Another approach is query generation, where natural language questions are transformed into formal query languages to retrieve relevant data from the table for answering (Zhong et al., 2017; Jin et al., 2022; Wang et al., 2020; Min et al., 2019). These approaches assumes that the table has a known, well-structured format suitable for query translation, but its effectiveness may be challenged when dealing with complex or unstructured tables. However, current researches still lack exploration of multi-turn dialogue. Our work is the first to explore the use of multi-turn dialogue for self-correction in the code generation process for TableQA.

3 Method

In this section, we introduce the system used, which addresses the Question Answering on Tabular Data task through an multi-turn interactive code generation approach, as shown in Fig. 2.

3.1 Information Retrieval and Prompt Construction

In this step, we construct suitable prompts based on the column names and example values of the table. Specifically, we retrieve the set of values for each column from the table and randomly select three values from this set as examples. The column names partially reflect the meaning of the columns, while the example values assist the model in better understanding the data types and content. For a given question, the model is required to first give the steps to solve the problem, then read the table data and perform calculations and analysis by writing Python code, and finally give the answer to the question in the form of a JSON dictionary. The prompt template is shown in Fig 3.

3.2 Data Augmentation

The original dataset provides only the final answers to the questions, lacking the reasoning process and code. To address this, we use GPT-4o to generate multi-turn dialogue data with chain-of-thought reasoning and code based on the constructed prompts, which is then used for subsequent model training.

Notably, the generated code may contain errors, such as accessing non-existent columns in the table or producing outputs that do not conform to the required format. To obtain as many correct code samples as possible while also equipping the model with error correction capabilities, we introduce a code executor and adopt a multi-turn interaction strategy. Specifically, upon receiving a model response, we check whether it contains Python code. If code is present, it is extracted and executed using the code executor. The execution results or traceback messages are then fed back to the model as dialogue messages, prompting it to modify the code accordingly or summarize the final answer. If no Python code is detected, the response is considered the final answer, and the dialogue terminates. The pseudo-code for this process is shown in Algorithm 1.

After obtaining the multi-turn interactive dialogue data, the model's answers are compared with the ground truth, and only the data with correct answers are retained for subsequent model training to avoid interference from low-quality data.

Algorithm 1: Multi-turn Interaction Strategy

```
Input: Question $\mathcal{Q}$; Maximum Number of
       Interactions $\mathcal{N}$.
Output: Dialogue Messages List $\mathcal{M}$.
$\mathcal{M} \leftarrow [(\text{User}, \mathcal{Q})]$;
for $n \in \{1, \cdots, \mathcal{N}\}$ do
    ▷ Obtain the response $\mathcal{R}$ from $LLM$.;
    $\mathcal{R} \leftarrow LLM(M)$;
    $\mathcal{M}.append((\text{Assistant}, \mathcal{R}))$;
    ▷ Extract the code from $\mathcal{R}$;
    $\mathcal{C} \leftarrow GetCode(R)$;
    if $\mathcal{C}$ is not None then
        ▷ Execute $\mathcal{C}$ and obtains output $\mathcal{O}$
        $\mathcal{O} \leftarrow Executor(\mathcal{C})$;
        $\mathcal{M}.append((\text{User}, \mathcal{O}))$;
    else
        break;
    end
end
```

3.3 Model Training

In this step, we fine-tune Qwen2.5-Coder-7B-Instruct (Qwen et al., 2025) using the augmented multi-turn dialogue data. Unlike single-turn dialogue data, multi-turn dialogue data contains multiple messages from the user or code executor. We use language modeling loss as the training loss

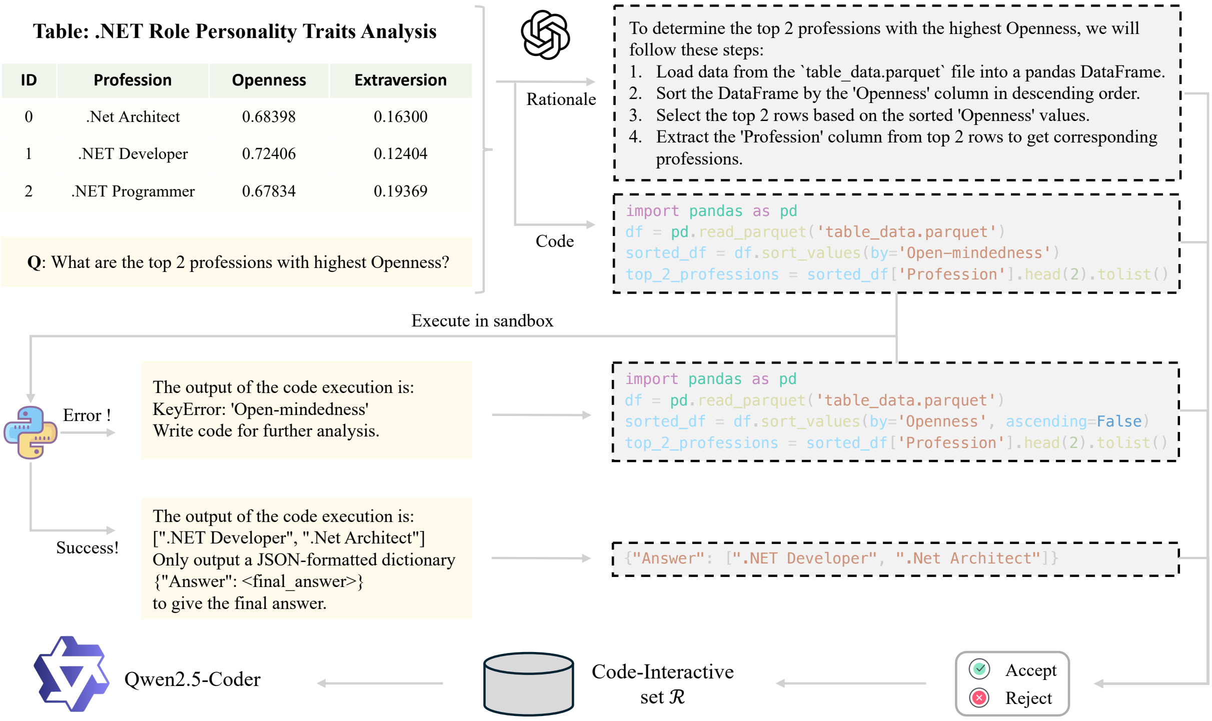


Table: .NET Role Personality Traits Analysis

ID	Profession	Openness	Extraversion
0	.Net Architect	0.68398	0.16300
1	.NET Developer	0.72406	0.12404
2	.NET Programmer	0.67834	0.19369

Q: What are the top 2 professions with highest Openness?

To determine the top 2 professions with the highest Openness, we will follow these steps:

1. Load data from the `table_data.parquet` file into a pandas DataFrame.
2. Sort the DataFrame by the 'Openness' column in descending order.
3. Select the top 2 rows based on the sorted 'Openness' values.
4. Extract the 'Profession' column from top 2 rows to get corresponding professions.

```
import pandas as pd
df = pd.read_parquet('table_data.parquet')
sorted_df = df.sort_values(by='Open-mindedness')
top_2_professions = sorted_df['Profession'].head(2).tolist()
```

The output of the code execution is:
KeyError: 'Open-mindedness'
Write code for further analysis.

```
import pandas as pd
df = pd.read_parquet('table_data.parquet')
sorted_df = df.sort_values(by='Openness', ascending=False)
top_2_professions = sorted_df['Profession'].head(2).tolist()
```

The output of the code execution is:
[".NET Developer", ".Net Architect"]
Only output a JSON-formatted dictionary
{"Answer": <final_answer>}
to give the final answer.

```
{"Answer": [".NET Developer", ".Net Architect"]}
```

(a) Training Process

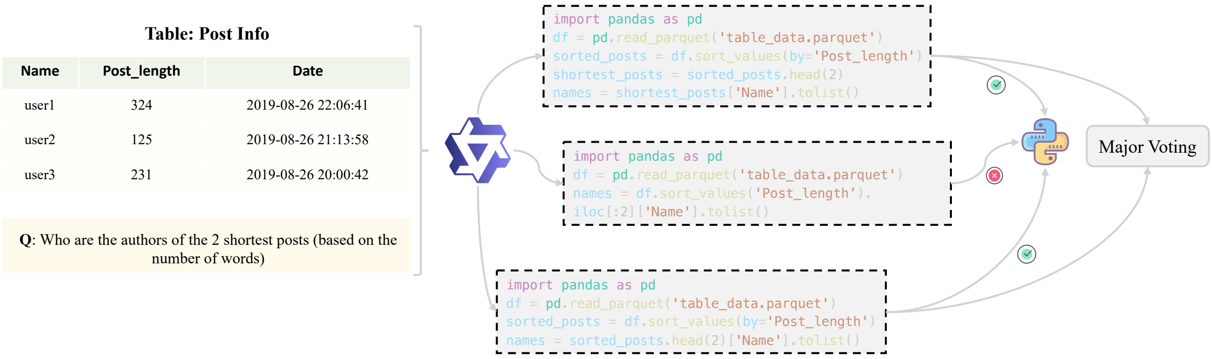


Table: Post Info

Name	Post_length	Date
user1	324	2019-08-26 22:06:41
user2	125	2019-08-26 21:13:58
user3	231	2019-08-26 20:00:42

Q: Who are the authors of the 2 shortest posts (based on the number of words)

```
import pandas as pd
df = pd.read_parquet('table_data.parquet')
sorted_posts = df.sort_values(by='Post_length')
shortest_posts = sorted_posts.head(2)
names = shortest_posts['Name'].tolist()
```

```
import pandas as pd
df = pd.read_parquet('table_data.parquet')
names = df.sort_values('Post_length’).
iloc[:2]['Name'].tolist()
```

```
import pandas as pd
df = pd.read_parquet('table_data.parquet')
sorted_posts = df.sort_values(by='Post_length')
names = sorted_posts.head(2)['Name'].tolist()
```

(b) Inference Process

Figure 2: Illustration of our proposed MICO.

function, calculating the loss only for the tokens generated by the model.

3.4 Model Inference

Similar to the data augmentation phase, during inference, the model interacts continuously with the code executor until the code analysis is complete and an answer is provided, or the maximum number of interactions is reached. If the maximum number of interactions is reached, the answer for the corresponding question will be set to null. For the decoding strategy, we first use greedy decoding for inference. For samples where the maximum interaction count is reached or the answer format is incorrect, we employ self-consistency (Wang et al., 2023) strategy. This involves sampling k responses and performing majority vote on the results to obtain final answer.

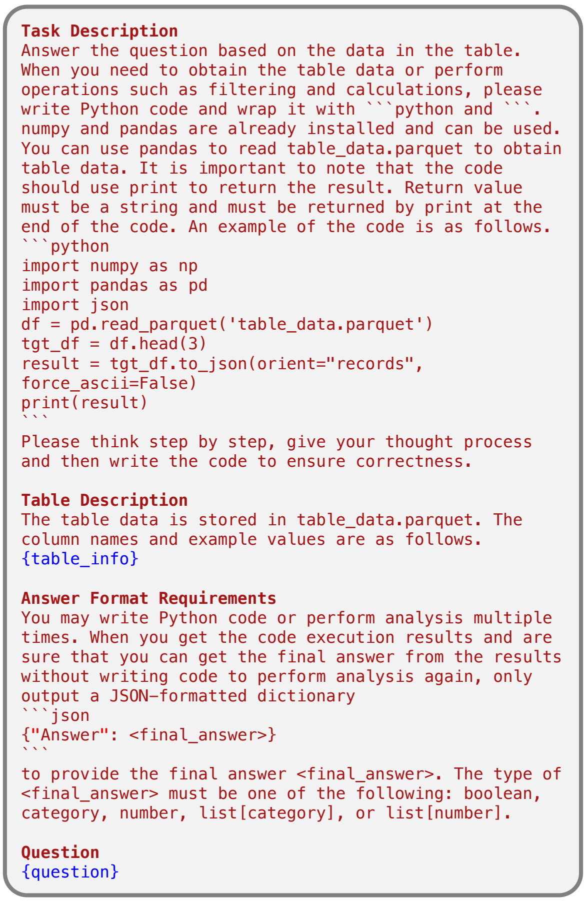
4 Experiments

4.1 Experimental Settings

We follow the exact dataset split as the competition organizers. The training set consists of 988 samples from 49 datasets, the validation set includes 320 samples from 16 datasets, and the test set contains 522 samples from 15 datasets. We use the databench_eval toolkit[2] to compute answer accuracy for evaluating different methods. For training, we fine-tune the model using LoRA (Hu et al.,

---

[2] https://github.com/jorses/databench_eval



~~~
Task Description
Answer the question based on the data in the table.
When you need to obtain the table data or perform
operations such as filtering and calculations, please
write Python code and wrap it with ```python and ```.
numpy and pandas are already installed and can be used.
You can use pandas to read table_data.parquet to obtain
table data. It is important to note that the code
should use print to return the result. Return value
must be a string and must be returned by print at the
end of the code. An example of the code is as follows.
```python
import numpy as np
import pandas as pd
import json
df = pd.read_parquet('table_data.parquet')
tgt_df = df.head(3)
result = tgt_df.to_json(orient="records",
force_ascii=False)
print(result)
~~~
Please think step by step, give your thought process
and then write the code to ensure correctness.

Table Description
The table data is stored in table_data.parquet. The
column names and example values are as follows.
{table_info}

Answer Format Requirements
You may write Python code or perform analysis multiple
times. When you get the code execution results and are
sure that you can get the final answer from the results
without writing code to perform analysis again, only
output a JSON-formatted dictionary
```json
{"Answer": <final_answer>}
```
to provide the final answer <final_answer>. The type of
<final_answer> must be one of the following: boolean,
category, number, list[category], or list[number].

Question
{question}
```

Figure 3: The prompt template.

2022). For LoRA parameters, we set the rank to 8, alpha to 16, and dropout to 0.1. The learning rate is set to 1e-4, with a batch size of 8. The model checkpoint with the lowest validation loss is selected for testing. During testing, the sampling temperature is set to 0.8, with a total of 10 sampled outputs per query.

4.2 Comparison Methods

For evaluating the effectiveness of our proposed system, we compare several methods: **Qwen-Single** utilizes Qwen2.5-Coder-7B-Instruct (Qwen et al., 2025) with greedy decoding, requiring the model to generate code that directly answers the question in a single response. **Qwen-Multi** follows a similar approach but uses the multi-turn interaction strategy to refine the response. **FullFT** builds on Qwen-Multi by fine-tuning the model with all generated multi-turn dialogue data. **FilterFT** further improves this by filtering the training data based on answer correctness, using only the filtered dataset for fine-tuning.

Method	# Score Databench
baseline	26.00
Qwen-Single	30.65
Qwen-Multi	52.30
FullFT	77.20
FilterFT	<u>81.03</u>
MICO	**82.18**

Table 1: The accuracy (%) of different methods. The results are presented such that the highest performance is denoted in bold, and the second highest performance is underlined.

5 Results

5.1 Comparison Results

Table 1 presents the comparison results of different models. Compared to Qwen-Multi, Qwen-Single shows a significant performance drop, indicating that providing both correctly formatted and accurate answers in a single-turn interaction is challenging. Introducing a multi-turn interaction strategy enables the system to refine code or organize answers based on execution feedback, significantly improving overall performance. After fine-tuning with enhanced multi-turn dialogue data, FullFT achieved a 24.9% accuracy improvement. This demonstrates the crucial role of multi-turn interaction data in enhancing the model's capabilities. Furthermore, by filtering data based on answer correctness, FilterFT achieved better training results with fewer data, increasing accuracy from 77.20% to 81.03%, demonstrating the crucial role of high-quality data in model training. By ensuring that only accurate responses contribute to learning, the model avoids the negative impact of low-quality samples, leading to more efficient training and improved overall performance. Building upon FilterFT, MICO optimized the decoding strategy by exploring the answer in a larger space through self-consistency for samples that reached the interaction limit or had formatting errors during greedy decoding, correcting these cases and achieving an additional 1.15% performance improvement.

5.2 Leaderboard Results

Table 2 and Table 3 present the performance of the top 10 teams in Databench dataset and Databench Lite dataset, respectively.
```

| Rank | Team | # Score |
|---|---|---|
| 1 | TeleAI | 95.02 |
| 2 | SRPOL AIS | 89.66 |
| 3 | AILS-NTUA | 87.16 |
| 4 | HITSZ-HLT | 86.97 |
| 5 | null33 | 86.02 |
| 6 | SBU-NLP | 85.63 |
| 7 | Oseibrefo-Liang | 84.67 |
| 8 | ITU-NLP | 84.10 |
| 9 | grazh | 83.72 |
| 10 | Howard University-AI4PC | 81.42 |

Table 2: Top-10 score on Databench for Open Leaderboard.

| Rank | Team | # Score |
|---|---|---|
| 1 | TeleAI | 92.91 |
| 2 | SRPOL AIS | 86.59 |
| 3 | SBU-NLP | 86.02 |
| 4 | Oseibrefo-Liang | 86.02 |
| 5 | HITSZ-HLT | 85.82 |
| 6 | ITU-NLP | 85.06 |
| 7 | tabaqa_team | 84.87 |
| 8 | null33 | 84.48 |
| 9 | Howard University-AI4PC | 80.46 |
| 10 | QleverAnswering-PUCRS | 80.27 |

Table 3: Top-10 score on Databench Lite for Open Leaderboard.

## 6 Conclusion

In this paper, we introduced the MICO framework for TableQA, aiming to enhance the accuracy and efficiency of answering queries over tabular data. By utilizing code generation as a proxy task for TableQA and incorporating multi-turn dialogue for feedback and self-correction, MICO effectively addresses the challenges of processing large, complex datasets and performing precise numerical computations. Our experimental results, conducted on the DataBench and DataBench Lite datasets, demonstrated the effectiveness of the MICO framework, with our system achieving a commendable rank of 4th out of 38 participants on the DataBench leaderboard and 5th out of 38 on the DataBench Lite leaderboard.

## Limitations

Despite the promising results, our MICO framework has some limitations. The multi-turn interactive process, while enhancing self-correction, increases computational complexity and inference time, especially for more complex queries or larger datasets. Additionally, the model's performance is sensitive to the quality of training data, and noisy or incorrect data can impact its ability to generate accurate responses. Lastly, the framework relies on the structure and format of tabular data, which may limit its generalization across different domains or datasets. Further evaluation on diverse datasets is needed to assess its scalability.

## Acknowledgment

This work was partially supported by the National Natural Science Foundation of China 62176076, Natural Science Foundation of Guang Dong 2023A1515012922, the Shenzhen Foundational Research Funding JCYJ20220818102415032 and CIPSC-SMP-ZHIPU Large Model Cross-Disciplinary Fund ZPCG20241119405.

## References

Julian Eisenschlos, Syrine Krichene, and Thomas Müller. 2020. Understanding tables with intermediate pre-training. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 281–296, Online. Association for Computational Linguistics.

Giang, Nguyen, Ivan Brugere, Shubham Sharma, Sanjay Kariyappa, Anh Totti Nguyen, and Freddy Lecue. 2024. Interpretable llm-based table question answering. *Preprint*, arXiv:2412.12386.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Mengkang Hu, Haoyu Dong, Ping Luo, Shi Han, and Dongmei Zhang. 2024. KET-QA: A dataset for knowledge enhanced table question answering. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 9705–9719, Torino, Italia. ELRA and ICCL.

Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: Recent advances.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019. A discrete hard EM approach for weakly supervised question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2851–2864, Hong Kong, China. Association for Computational Linguistics.

Thomas Mueller, Francesco Piccinno, Peter Shaw, Massimo Nicosia, and Yasemin Altun. 2019. Answering conversational questions on structured data without logical forms. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5902–5910, Hong Kong, China. Association for Computational Linguistics.

Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, Mutethia Mutuma, Ben Rosand, Isabel Trindade, Renusree Bandaru, Jacob Cunningham, Caiming Xiong, Dragomir Radev, and Dragomir Radev. 2022. FeTaQA: Free-form table question answering. *Transactions of the Association for Computational Linguistics*, 10:35–49.

Jorge Osés-Grijalba, Luis Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2025. SemEval-2025 task 8: Question answering over tabular data. In *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, Vienna, Austria. Association for Computational Linguistics.

Vaishali Pal, Andrew Yates, Evangelos Kanoulas, and Maarten de Rijke. 2023. MultiTabQA: Generating tabular answers for multi-table question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6322–6334, Toronto, Canada. Association for Computational Linguistics.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, Haoze Li, Haoxuan Lan, Jiaming Tian, Jing Yuan, Junbo Zhao, Junlin Zhou, Kaizhe Shou, Liangyu Zha, Lin Long, Liyao Li, Pengzuo Wu, Qi Zhang, Qingyi Huang, Saisai Yang, Tao Zhang, Wentao Ye, Wufang Zhu, Xiaomeng Hu, Xijun Gu, Xinjie Sun, Xiang Li, Yuhang Yang, and Zhiqing Xiao. 2024. Tablegpt2: A large multimodal model with tabular data integration. *Preprint*, arXiv:2411.02059.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

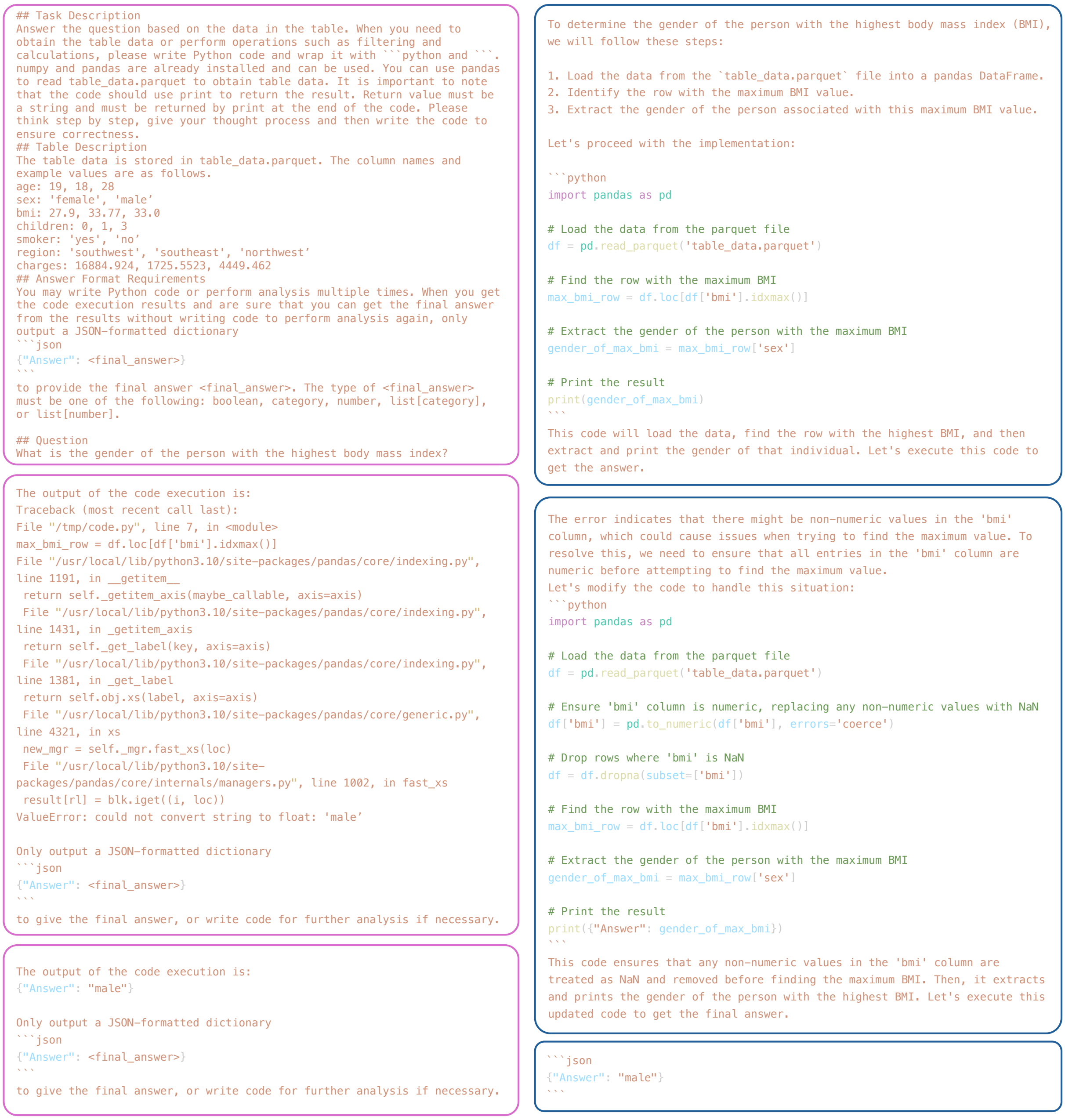
Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xinrun Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, Guanglin Niu, Tongliang Li, and Zhoujun Li. 2024. Tablebench: A comprehensive and complex benchmark for table question answering. *Preprint*, arXiv:2408.09174.

Yilun Zhao, Lyuhao Chen, Arman Cohan, and Chen Zhao. 2024. TaPERA: Enhancing faithfulness and interpretability in long-form table QA by content planning and execution-based reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12824–12840, Bangkok, Thailand. Association for Computational Linguistics.

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

Fan Zhou, Mengkang Hu, Haoyu Dong, Zhoujun Cheng, Fan Cheng, Shi Han, and Dongmei Zhang. 2022. TaCube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2278–2291, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

# A Case Study



~~~
## Task Description
Answer the question based on the data in the table. When you need to
obtain the table data or perform operations such as filtering and
calculations, please write Python code and wrap it with ```python and ```.
numpy and pandas are already installed and can be used. You can use pandas
to read table_data.parquet to obtain table data. It is important to note
that the code should use print to return the result. Return value must be
a string and must be returned by print at the end of the code. Please
think step by step, give your thought process and then write the code to
ensure correctness.
## Table Description
The table data is stored in table_data.parquet. The column names and
example values are as follows.
age: 19, 18, 28
sex: 'female', 'male’
bmi: 27.9, 33.77, 33.0
children: 0, 1, 3
smoker: 'yes', 'no’
region: 'southwest', 'southeast', 'northwest’
charges: 16884.924, 1725.5523, 4449.462
## Answer Format Requirements
You may write Python code or perform analysis multiple times. When you get
the code execution results and are sure that you can get the final answer
from the results without writing code to perform analysis again, only
output a JSON-formatted dictionary
```json
{"Answer": <final_answer>}
```
to provide the final answer <final_answer>. The type of <final_answer>
must be one of the following: boolean, category, number, list[category],
or list[number].

Question
What is the gender of the person with the highest body mass index?
~~~

~~~
To determine the gender of the person with the highest body mass index (BMI),
we will follow these steps:

1. Load the data from the `table_data.parquet` file into a pandas DataFrame.
2. Identify the row with the maximum BMI value.
3. Extract the gender of the person associated with this maximum BMI value.

Let's proceed with the implementation:

```python
import pandas as pd

# Load the data from the parquet file
df = pd.read_parquet('table_data.parquet')

# Find the row with the maximum BMI
max_bmi_row = df.loc[df['bmi'].idxmax()]

# Extract the gender of the person with the maximum BMI
gender_of_max_bmi = max_bmi_row['sex']

# Print the result
print(gender_of_max_bmi)
```
This code will load the data, find the row with the highest BMI, and then
extract and print the gender of that individual. Let's execute this code to
get the answer.
~~~

~~~
The output of the code execution is:
Traceback (most recent call last):
File "/tmp/code.py", line 7, in <module>
max_bmi_row = df.loc[df['bmi'].idxmax()]
File "/usr/local/lib/python3.10/site-packages/pandas/core/indexing.py",
line 1191, in __getitem__
 return self._getitem_axis(maybe_callable, axis=axis)
 File "/usr/local/lib/python3.10/site-packages/pandas/core/indexing.py",
line 1431, in _getitem_axis
 return self._get_label(key, axis=axis)
 File "/usr/local/lib/python3.10/site-packages/pandas/core/indexing.py",
line 1381, in _get_label
 return self.obj.xs(label, axis=axis)
 File "/usr/local/lib/python3.10/site-packages/pandas/core/generic.py",
line 4321, in xs
 new_mgr = self._mgr.fast_xs(loc)
 File "/usr/local/lib/python3.10/site-
packages/pandas/core/internals/managers.py", line 1002, in fast_xs
 result[rl] = blk.iget((i, loc))
ValueError: could not convert string to float: 'male’

Only output a JSON-formatted dictionary
```json
{"Answer": <final_answer>}
```
to give the final answer, or write code for further analysis if necessary.
~~~

~~~
The error indicates that there might be non-numeric values in the 'bmi'
column, which could cause issues when trying to find the maximum value. To
resolve this, we need to ensure that all entries in the 'bmi' column are
numeric before attempting to find the maximum value.
Let's modify the code to handle this situation:
```python
import pandas as pd

# Load the data from the parquet file
df = pd.read_parquet('table_data.parquet')

# Ensure 'bmi' column is numeric, replacing any non-numeric values with NaN
df['bmi'] = pd.to_numeric(df['bmi'], errors='coerce')

# Drop rows where 'bmi' is NaN
df = df.dropna(subset=['bmi'])

# Find the row with the maximum BMI
max_bmi_row = df.loc[df['bmi'].idxmax()]

# Extract the gender of the person with the maximum BMI
gender_of_max_bmi = max_bmi_row['sex']

# Print the result
print({"Answer": gender_of_max_bmi})
```
This code ensures that any non-numeric values in the 'bmi' column are
treated as NaN and removed before finding the maximum BMI. Then, it extracts
and prints the gender of the person with the highest BMI. Let's execute this
updated code to get the final answer.
~~~

~~~
The output of the code execution is:
{"Answer": "male"}

Only output a JSON-formatted dictionary
```json
{"Answer": <final_answer>}
```
to give the final answer, or write code for further analysis if necessary.
~~~

~~~
```json
{"Answer": "male"}
```
~~~

Figure 4: Case study.