LyS at SemEval 2025 Task 8: Zero-Shot Code Generation for Tabular QA

Adrián Gude
adrian.lopez.gude@udc.es

Roi Santos-Ríos
roi.santos.rios@udc.es

Francisco Prado-Valiño
francisco.prado.valino@udc.es

Ana Ezquerro
ana.ezquerro@udc.es

Jesús Vilares
jesus.vilares@udc.es

Universidade da Coruña, CITIC
Departamento de Ciencias de la Computación y Tecnologías de la Información
Campus de Elviña s/n, 15071, A Coruña, Spain

Abstract

This paper describes our participation in SemEval 2025 Task 8, focused on Tabular Question Answering. We developed a zero-shot pipeline that leverages an Large Language Model to generate functional code capable of extracting the relevant information from tabular data based on an input question. Our approach consists of a modular pipeline where the main code generator module is supported by additional components that identify the most relevant columns and analyze their data types to improve extraction accuracy. In the event that the generated code fails, an iterative refinement process is triggered, incorporating the error feedback into a new generation prompt to enhance robustness. Our results show that zero-shot code generation is a valid approach for Tabular QA, achieving rank 33 of 53 in the test phase despite the lack of task-specific fine-tuning.

1 Introduction

Tabular Question Answering (Tabular QA) has huge potential in real-world applications such as financial analysis, business intelligence, and scientific data exploration, where structured databases serve as the primary source of information. Unlike traditional text-based Question Answering (QA), which primarily deals with unstructured data, Tabular QA requires extracting information from structured tables to be able to answer the input questions, thus involving reasoning about diverse table schemas, column relationships, and heterogeneous data types.

Complex supervised systems have been proposed to deal with the structured nature of Tabular QA, either leveraging structured prediction with language representations (Herzig et al., 2020; Yin et al., 2020) or by formulating the task as a sequence-to-sequence problem (Zhong et al., 2017; Yu et al., 2018; Pal et al., 2023). However, with the rise of instruction-based Large Language Models (LLM) (Brown et al., 2020), recent approaches have shifted away from reliance on large annotated datasets, instead reframing the task as a zero-shot generation problem (Cao et al., 2023).

In this work, we further explore instruction-based LLMs to dynamically generate code functions capable of retrieving relevant data from tables based on the input question in a zero-shot manner. To enhance accuracy and reliability, we developed a modular three-staged pipeline that includes: (i) a column selection mechanism to determine the most relevant columns and their data-type, (ii) a code generation module responsible for producing executable code and (iii) an iterative error handling module that, in case the initial code execution fails, tries to fix the generated code accordingly.

Our group tested this approach within the SemEval 2025 Task 8 event (Osés Grijalba et al., 2025), which provided a diverse dataset featuring real-world tabular data.[1] The competition required models to produce answers in multiple formats, including boolean, categorical, numerical, and list-based outputs. Our model was designed to generalize across different table structures, making it adaptable to various datasets beyond the shared task, ensuring robustness and broad applicability. Although our approach demonstrated strong performance in code generation and execution, subsequent analysis revealed that the model struggles with columns containing complex data types (lists, dictionaries, etc.) and ambiguous queries, particularly for list-based responses.

2 Background

Question Answering (QA) has been gaining significant attention in recent years, driven by the need for models capable of reasoning over structured data.

[1] Our implementation is fully available at https://github.com/adrian-gude/Tabular_QA (Feb. 2025).

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 1282–1288

Early tasks in QA mainly focused on retrieving information from unstructured text sources (Rajpurkar et al., 2016; Yang et al., 2018), but the increasing availability of structured datasets has led to new challenges in understanding and querying tabular data. Unlike classic text-based QA, where answers are retrieved from free-form text, Tabular QA requires a higher level of interpretation and robustness to map questions to relevant columns and rows, handle missing values, and compute statistics when necessary.

In parallel, several datasets have been introduced to benchmark Tabular QA models, including WikiTableQuestions (Pasupat and Liang, 2015), SQA (Iyyer et al., 2017), and the more recent DataBench dataset (Osés Grijalba et al., 2024), which provides real-world tabular data for evaluating models in different scenarios.

Structured Tabular QA Most state-of-the-art approaches for Tabular QA leverage a pretrained language model —equipped with an specialized encoding module to represent tabular information— tailored for structured prediction. For example, TAPAS (Herzig et al., 2020) feeds both the input question and the flattened table into BERT (Devlin et al., 2019) as a single sequence, and finetunes the architecture to select relevant columns and predict an aggregation function. Similarly, TACUBE (Zhou et al., 2022) combines a cube constructor with BART (Lewis et al., 2020) to predict the real answers based on the input question and the results of the cube operations.

Generative Tabular QA To address the rigidity of structured approaches, recent works have explored generative models for program synthesis, where an LLM is finetuned to generate executable programs or instructions (in the form of SQL queries, for example) to be applied against tabular sources. Zhong et al. (2017) proposed SEQ2SQL, a sequence-to-sequence model to translate natural language into SQL syntax, incorporating query-space pruning to significantly simplify and enhance the generative task. Later, Yin et al. (2020) joined both concepts by optimizing tabular embeddings that fit both generative and structured purposes.

Zero-Shot Code Generation More recently, advancements in code generation have enabled a paradigm shift in Tabular QA, driven by powerful multipurpose LLMs with strong coding capabilities, such as Qwen (Bai et al., 2023) and Mistral's Codestral (Jiang et al., 2023). These models facilitate a zero-shot approach to program synthesis, eliminating the need for predefined templates or large annotated datasets. Instead, zero-shot generation allows the system to dynamically adapt to different schemes without explicit prior knowledge of the table structure (Cao et al., 2023), thus providing flexibility and scalability.

Despite its potential, zero-shot code generation models still face big challenges, particularly in error handling, runtime execution failures, and schema variability. Building on this approach, our work extends an instruction-based model with error awareness, enabling it to detect and recover from execution failures in an iterative error-recovery mechanism, where the model dynamically analyzes execution failures and regenerates code based on error feedback.

3 System Overview

Our approach for the SemEval 2025 Task 8 iterates upon the code generation approaches for Tabular QA, where the core component is a pretrained LLM responsible of generating executable code to extract the answer from the tables. To build upon prior works (Herzig et al., 2020), we incorporated a module that helps selecting the columns relevant to the question, while also identifying the data types of their content. Moreover, we incorporate an error-fixing module that attempts to catch runtime errors and integrates them as part of a new prompt, guiding the LLM to refine its code generation.

Figure 1 shows an schematic view of the architecture of our system. We have designed a modular pipeline that features three main components, which we describe below: (i) a column selector, (ii) an answer generator and (iii) a code fixer.

Column Selector Instead of relying on manually crafted heuristics or embedding similarity measures, the first component of our system leverages an instruction-based LLM tasked to identify the most relevant columns of a tabular source from an input question in natural language form. Our template provides the list of column names and instructs the model to return only those that are essential for answering the query.[2]

[2]All our prompts are available in the code publicly available at GitHub.

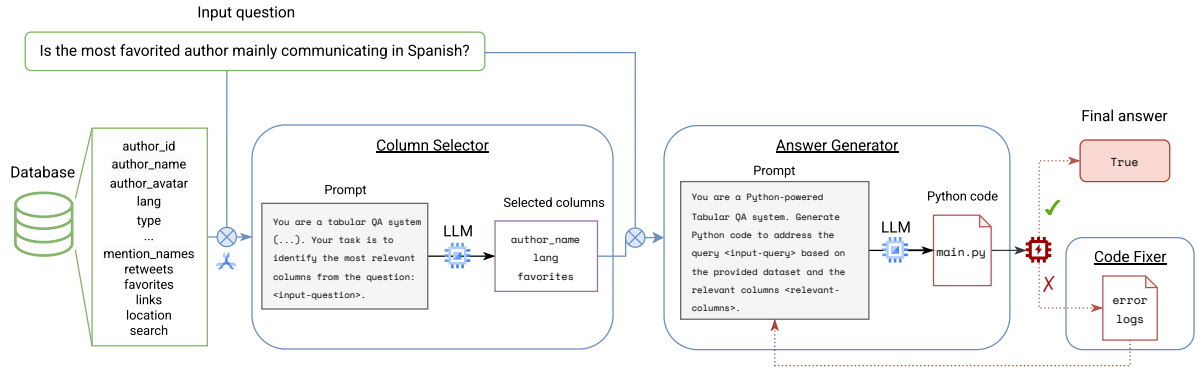

Figure 1: Architecture of our system. Different symbols are used to represent different elements of our pipeline: ⊗ merges information in a prompting-like form, represents a preprocessing step, indicates LLM inference (with optional post-processing steps), and runs Python code and catches error logs. Solid lines are used to indicate fixed pipeline steps while dotted lines indicate optional steps that are executed depending on partial results of the system. Green boxes represent elements provided in the task.

Answer Generator Once the relevant columns are identified, the second component of our pipeline is instructed to generate executable code that retrieves the answers from the tabular source using both the input query and the relevant columns extracted in the previous step. As part of our prompt, we guided the LLM to generate Python programming code and postprocessed the output to ensure that only Python lines were passed throught the next module. Python language was chosen since it is widely used in data analysis and has extensive support for tabular data processing through libraries such as Pandas.

Code Fixer The final component of our pipeline captures execution errors that might occur due to incorrect syntax, schema mismatches, or runtime exceptions. This module captures the error messages and re-generates a corrected function by feeding the error context back into the LLM. To achieve this, we used a structured prompt that includes the code that causes an error with the corresponding error description.

Preprocessing Since our system strongly relies on a well-formatted prompt, we manually designed a preprocessing step to ensure a consistent format to feed our system. We standardized column names for simplified versions (removing emoji and all non-alphanumerical characters except punctuation symbols) to prevent possible errors in the Answer Generator caused by mismatches between the table structure and the generated code. We identified enum-like column types, such as the case of categorical attributes with a finite amount of strings as a value (e.g. a "*Survey*" column that only contains "*Yes*", "*No*" or "*Maybe*"), and inferred a common scheme so to ensure consistency across different attributes, thus reducing errors related to unexpected variations in categorical values.

4 Experimental Setup

Our system relies on open-source LLMs for zero-shot code generation. This way, no explicit training nor finetuning was conducted. Instead, we used the available training phase datasets to validate different LLMs and select the best performing one for the final test phase.

Dataset The dataset provided for the task is divided into three sets: *training*, *development* (aka *dev*), and *test*. In our case, since we had opted for a zero-shot approach, the training set remained unused during the development phase, using only the dev set for our experiments. During this stage we tried different LLMs to compare their ability to generate the adequate Python code to answer the input questions. To do that, we analyzed the accuracy obtained with respect to the ground truth of the validation set, together with manual checks to assess the quality of the generated code.

Evaluation The official evaluation consists of two subtasks, where *Subtask 1* uses all available data sources to answer the input question, while *Subtask 2* operates on a limited database, sampling a maximum of 20 rows per table to perform queries.

System Setup We conducted experiments with different open-source LLMs adjusted to our hardware limitations, specifically pretrained for

instruction-based code generation: Qwen-2.5-Coder (Bai et al., 2023) (with 7B and 32B versions), Mistral-7B and Codestral-22B —the later two from Mistral (Jiang et al., 2023).

To run the generated code we relied on Python 3.10.12 with Pandas 2.2.3 as a requirement. Due to VRAM constraints, all models were executed with 4-bit quantization, using a greedy generation strategy with a temperature of 0.7.

5 Analysis of Results

In this section, we present the evaluation of our system on the task. We first report performance during the development phase (§5.1), where we experimented with different models on the validation dataset, followed by the final test phase (§5.2), where our system was evaluated on the test dataset through CodaBench submissions.[3]

5.1 Development Phase

As explained before, during the development phase we focused on selecting the best performing LLM just using the dev set; that is, dismissing the training set. At this first stage, our pipeline was conformed by only the Answer Generator module.

The results obtained for this original setup, presented in Table 1, show that larger models such as Qwen-2.5-Coder^{32B} significantly outperform smaller models, with accuracy gains of over 20 points compared to Qwen2.5-Coder^{7B}. Regardless of the selected model, our zero-shot approach consistently outperforms the baseline system (Osés Grijalba et al., 2025) in both subtasks. Evaluation metrics indicate higher scores for Subtask 2 than for Subtask 1, likely due to the smaller input size, which reduces the amount of information introduced in the prompt and minimizes potential ambiguities when execution the generated code. We also notice a performance drop when breaking down the accuracy by the datatype, where even the best LLM struggles when generating answers for categorical list-like attributes.

Ablation Study We relied on the results displayed in Table 1 to select the best performing LLM, which served as the foundation for integrating the additional modules that could further enhance performance (see Figure 1). Table 2 shows the results when varying the components of the pipeline while maintaining Qwen-2.5-Coder^{32B} as backbone. The AG (Answer Generator only) setup corresponds to the result displayed in Table 1, from which the extra components of our pipeline where compared to see if there was an actual improvement when introducing error-awareness and column preselection. The AG+CS (AG with Column Selector) setup shows a clear improvement of 3 and 2 points in each subtask with respect to the AG-only model, outlining the importance of first asking the LLM to filter the relevance of the input attributes. Lastly, when integrating the Code Fixer (CF) with an enhanced column selection (ECS) to feed richer information about feature variations to the prompt, our final system setup (AG+ECS+CF) maintains almost the same performance over Subtask 2 but improves 7 points in Subtask 1, proving that integrating error feedback to the model assists the LLM for better querying larger databases. Specifically, the largest performance boost is obtained in categorical list-like attributes, where the accuracy increases 10 points with respect to the AG+CS model.

5.2 Final Test Phase

The best performing configuration (AG+ECS+CF) was selected to participate in the competition. Our zero-shot approach reached 65 points of accuracy in Subtask 1 and 68 points in Subtask 2. So, we ranked in the 32th (Subtask 1) and 31th (Subtask 2) positions out of 49 participants in the *General* category, and 23th (Subtask 1) and 21th (Subtask 2) positions out of 35 participants in the *Open models* category.

Our results during the development phase (84 and 85 points for Subtasks 1 and 2, respectively) suffered a significant drop of 20 points (approx.) in accuracy with respect to the validation results, likely due to the greater complexity of datatypes presented in the test tables. For instance, the test set presents multiple columns with lists that are not enclosed by square brackets, or that have variable separators for their elements (commas or semicolons); and dictionaries with a variable amount of keys.[4] Tables 1 and 2 show a clear difference in terms of accuracy when considering more complex datatypes: boolean accuracy reaches more than 80 points, while list-like types do not surpass 75 points. This might indicate that the LLM is not able to infer these complex schemes on the test

[3] https://www.codabench.org/competitions/3360/.

[4] For example, a cell of the form: `Education;Social Protection;Agriculture, Fishing and Forestry` or `{'service': 5.0, 'cleanliness': 5.0, 'overall': 5.0, 'value': 4.0, 'location': 5.0}`.

		boolean	category	number	list[category]	list[number]	μ	β
S1	Qwen-2.5-Coder^{7B}	67.19	68.75	75.00	3.12	3.12	43.44	27.00
	Mistral^{7B}	51.56	59.37	73.44	35.94	34.37	50.94	
	Codestral^{22B}	73.44	**82.81**	**82.81**	48.44	48.44	67.19	
	Qwen-2.5-Coder^{32B}	**81.25**	78.12	75.00	**65.62**	**70.31**	**74.06**	
S2	Qwen-2.5-Coder^{7B}	81.25	84.37	85.93	6.25	1.56	51.87	26.00
	Mistral^{7B}	46.87	56.25	65.62	32.81	25.00	45.31	
	Codestral^{22B}	71.87	**89.06**	84.37	53.12	60.94	71.87	
	Qwen-2.5-Coder^{32B}	**84.37**	**89.06**	**85.94**	**75.00**	**75.00**	**81.87**	

Table 1: Performance of different LLMs on the validation set for Subtasks 1 and 2 (*S1* and *S2*, respectively), where the pipeline only contains the Answer Generator module. Columns μ and β indicate the average and baseline performance, respectively. The best performance is highlighted in bold.

		boolean	category	number	list[category]	list[number]	μ
S1	AG	81.25	78.12	75.00	65.62	70.31	74.06
	AG+CS	82.81	78.12	78.12	68.75	79.69	77.50
	AG+ECS+CF	**89.06**	**85.94**	**85.94**	**78.12**	**85.94**	**85.00**
S2	AG	84.37	**89.06**	85.94	75.00	75.00	81.87
	AG+CS	84.37	**89.06**	**90.62**	73.44	**79.69**	83.44
	AG+ECS+CF	**89.06**	**89.06**	**90.62**	**76.56**	78.12	**84.69**

Table 2: Performance on the validation set for Subtasks 1 and 2 (*S1* and *S2*, respectively) when integrating different components of the pipeline with Qwen-2.5-Coder^{32B} as backbone. The best performance is highlighted in bold.

set, producing errors that are propagated from the Column Selector module to the Answer Generator.

6 Conclusions and Future Work

In this work we propose a zero-shot approach for Tabular QA that demonstrated a strong performance for the SemEval 2025 Task 8, ranking among the best systems in the development phase, although suffering from a performance drop in the test phase. Still, our system shows that an instruction-based approach allows to dynamically adapt to different dataset schemes without requiring additional training or finetuning, surpassing the baseline model even with limited hardware resources available.

Future work will focus on further refining prompt templates, improving schema adaptation, optimizing execution efficiency or incorporating a voting system with different LLMs. Improving the detection of these complex datatypes is also critical, as they allow the model to answer questions on less structured tables —which constitute the majority of online data—, ultimately making the system more generalizable.

Hardware Setup

Our hardware resources are somewhat limited by today's standards. We had shared access to an Intel Core i9-10920X at 3.50 GHz with 258 GiB RAM and two integrated NVIDIA RTX 3090, so we opted to perform zero-shot instead of finetuning the LLMs.

Acknowledgments

We acknowledge grants SCANNER-UDC (PID2020-113230RB-C21) funded by MICIU/AEI/10.13039/501100011033; GAP (PID2022-139308OA-I00) funded by MICIU/AEI/10.13039/501100011033/ and ERDF, EU; LATCHING (PID2023-147129OB-C21) funded by MICIU/AEI/10.13039/501100011033 and ERDF, EU; CIDMEFEO funded by the Spanish National Statistics Institute (INE); as well as funding by Xunta de Galicia (ED431C 2024/02), and Centro de Investigación de Galicia "CITIC", funded by the *Xunta de Galicia* through the collaboration agreement between the *Consellería de Cultura, Educación, Formación Profesional e Universidades* and the Galician universities for the reinforcement of the research centres of the Galician University System (CIGUS).

CITIC, as a center accredited for excellence within the Galician University System and a member of the CIGUS Network, receives subsidies from the Department of Education, Science, Universities, and Vocational Training of the Xunta de Galicia. Additionally, it is co-financed by the EU through the FEDER Galicia 2021-27 operational program (Ref.ED431G 2023/01)

References

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Yihan Cao, Shuyi Chen, Ryan Liu, Zhiruo Wang, and Daniel Fried. 2023. API-Assisted Code Generation for Question Answering on Varied Table Structures. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14536–14548, Singapore. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333, Online. Association for Computational Linguistics.

Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. 2017. Search-based Neural Structured Learning for Sequential Question Answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1821–1831, Vancouver, Canada. Association for Computational Linguistics.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *Preprint*, arXiv:2310.06825.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Jorge Osés Grijalba, L. Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2024. Question Answering over Tabular Data with DataBench: A Large-Scale Empirical Evaluation of LLMs. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13471–13488, Torino, Italia. ELRA and ICCL.

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.

Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ Questions for Machine Comprehension of Text. *Preprint*, arXiv:1606.05250.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages

2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online. Association for Computational Linguistics.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594, New Orleans, Louisiana. 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 Prompts used

A.1 Answer Generator

```
Role and Context
You are a Python-powered Tabular Data Question-Answering
      System. Your core expertise lies in understanding
      tabular datasets and crafting Python scripts to
      generate precise solutions to user queries.

Task Description:
Generate Python code to address a query based on the
      provided dataset. The output must:

- Use the dataset and query as given, avoiding any external
      assumptions.
- Adhere to strict syntax rules for Python, ensuring the
      code runs flawlessly without external modifications.
- Retain the original column names of the dataset in your
      script.

Input Specification
    dataset: A Pandas DataFrame containing the data to be
          analyzed.
    question: A string outlining the specific query.

Output Specification
    Return only the Python code that solves the query in the
          function, excluding any introductory explanations
          or comments. The function must:
        Include all essential imports.
        Be concise and functional, ensuring the script can
              be executed without additional modifications.
        Use the dataset and return a result of type number,
              categorical value, boolean value, or a list of
              values.

Code Template
    Below is a reusable code structure for reference:
    Return only the code inside the function, without any
          outer indentation.
    Complete the function with your solution, ensuring the
          code is functional and concise.

import pandas as pd
def answer(df: pd.DataFrame) -> None:
    df.columns = {list(df.columns)} # Retain original column
          names
    # The columns used in the solution : {selected_columns}
    {columns_unique}
    # Your solution goes here
    ...
    >>>{row["question"]}
```

A.2 Column Selector

```
You are a tabular QA system specialized in understanding and
      analyzing datasets. Your task is to identify the most
      relevant columns from a given dataset that can answer a
      specific question.

You will be provided with a list of column names from the
      dataset.
Based on the question, analyze the provided column names and
      determine which ones are likely to contain the
      information required to answer the question. You only
      have to answer the question based on the provided
      column names in the formmating described below.

Input Format:
    column_names: A list of column names from the dataset.
          Each column name is enclosed in single quotes and
          separated by commas. The column names may contain
          spaces and special characters.
    question: A string containing the question to be
          answered.

Output Format:
    A list of the relevant column names. The output should
          be a subset of the provided column names. Maintain
          the names EXACTLY as provided, special characters
          and all, for example < or >. If no columns are
          relevant, return an empty list.
    Only the relevant column names should be returned in
          list format, without any additional information or
          formatting.

Example:
    column_names: ['Name', 'Age', 'Email', 'Purchase Date',
          'Product']
    question: 'Which product was purchased?'
    Output: ['Product']

Input:
    column_names: {column_names}
    question: {question}
```

A.3 Code Fixer

```
Role and Context
You are a Python-powered Tabular Data Question-Answering
      System. Your core expertise lies in understanding
      tabular datasets and crafting Python scripts to
      generate precise solutions to user queries.

Task Description:
Fix the Python code to address a query based on the provided
      dataset. The output must:

- Use the dataset and query as given, avoiding any external
      assumptions.
- Adhere to strict syntax rules for Python, ensuring the
      code runs flawlessly without external modifications.
- Retain the original column names of the dataset in your
      script.

Input Specification
    code: The Python code that needs to be fixed.
    error: The error message that results from running the
          code.

Output Specification
    Return only the Python code that solves the query in the
          function, excluding any introductory explanations
          or comments. The function must:
        Include all essential imports.
        Be concise and functional, ensuring the script can
              be executed without additional modifications.
        Use the dataset and return a result of type number,
              categorical value, boolean value, or a list of
              values.
Code:
    Below is the piece of code that needs to be fixed, along
          with the error message that results from running
          the code:
    {response}

Error: {error}
```