LyS at IberLEF 2025 Task PRESTA: Zero-Shot Code Generation for Spanish Tabular QA

Roi Santos-Ríos, Adrián Gude, Francisco Prado-Valiño, Ana Ezquerro and Jesús Vilares

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 PRESTA, an IberLEF 2025 task, 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 4 out of 7 in the test phase despite the lack of task-specific fine-tuning.

Keywords

Tabular Question Answering, Large Language Models, Zero-Shot, Code Generation

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 [1, 2] or by formulating the task as a sequence-to-sequence problem [3, 4, 5]. However, with the rise of instruction-based Large Language Models (LLM) [6], recent approaches have shifted away from reliance on large annotated datasets, instead reframing the task as a zero-shot generation problem [7].

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 IberLEF's [8] PRESTA task [9], 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

IberLEF 2025, September 2025, Zaragoza, Spain

roi.santos.rios@udc.es (R. Santos-Ríos); adrian.lopez.gude@udc.es (A. Gude); francisco.prado.valino@udc.es (F. Prado-Valiño); ana.ezquerro@udc.es (A. Ezquerro); jesus.vilares@udc.es (J. Vilares)

https://dunque.github.io/ (R. Santos-Ríos); https://adrian-gude.github.io/ (A. Gude); https://anaezquerro.github.io/ (A. Ezquerro); https://www.grupolys.org/~jvilares/ (J. Vilares)

0009-0009-0541-1232 (R. Santos-Ríos); 0009-0005-7215-3940 (A. Gude); 0009-0004-3543-2564 (F. Prado-Valiño); 0009-0006-2347-9706 (A. Ezquerro); 0000-0003-2941-1834 (J. Vilares)

[1] Our implementation is fully available at https://github.com/Dunque/Tabular_QA_es (May. 2025).

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. Early tasks in QA mainly focused on retrieving information from unstructured text sources [10, 11], 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 [12], SQA [13], and the more recent DataBenchSPA dataset [14], 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 [1] feeds both the input question and the flattened table into BERT [15] as a single sequence, and finetunes the architecture to select relevant columns and predict an aggregation function. Similarly, TaCube [16] combines a cube constructor with BART [17] 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. [3] 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. [2] 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 [18] and Mistral's Codestral [19]. 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 [7], 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 PRESTA task 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 [1], 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.

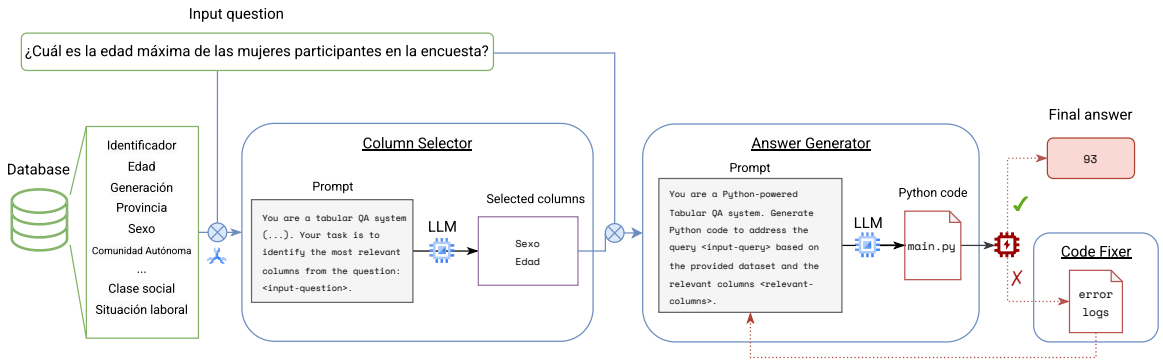

Figure 1: Architecture of our system. Different symbols are used to represent different elements of our pipeline: ⊗ merges information in a prompting-like form, represents a preprocessing step, indicates LLM inference (with optional post-processing steps), and runs Python code and catches error logs. 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.

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]

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

[2]All our prompts are available in the code publicly available at GitHub, as well as in the Annex section.

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 checking if the system is able to retrieve the answer from the tables, comparing its output with the expected one. Still, in this competition the evaluation scripts allow for non-meaningful differences in the outputs; i.e. the system outputs 125.0 (float), and the expected result is 125 (int). In this case it is counted as a correct response.

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 [18] (with 7B and 32B versions), Mistral-7B and Codestral-22B —the later two from Mistral [19].

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 the only model able to outperform the baseline system [20] is Qwen-2.5-Coder^{32B}.

The majority of code answers outputted from the Qwen-2.5-Coder^{7B}, Mistral^{7B} and Codestral^{22B} models stem from trying to use the function "split()", which cannot be used with the majority of datatypes present in the columns of the dataset.

The smaller LLM models needed more postprocessing in order to be able to extract the code they generated from the rest of the response. They usually add unnecessary textual descriptions of the code, even though it is stated in the prompt that none of that is necessary, and will make the execution fail.

[3] https://www.codabench.org/competitions/5538/.

		boolean	category	number	list[category]	list[number]	μ	β
Valid.	Qwen-2.5-Coder^{7B}	20.00	45.45	27.27	16.67	05.56	24.00	49.00
	Mistral^{7B}	00.00	00.00	09.09	00.00	00.00	02.00	
	Codestral^{22B}	50.00	27.27	36.36	27.78	38.89	36.00	
	Qwen-2.5-Coder^{32B}	**60.00**	**63.64**	**45.45**	**66.67**	**72.22**	**61.00**	
Test	Qwen-2.5-Coder^{7B}	00.00	00.00	45.00	05.00	10.00	12.00	49.00
	Mistral^{7B}	00.00	05.00	45.00	00.00	10.00	12.00	
	Codestral^{22B}	35.00	20.00	50.00	45.00	40.00	38.00	
	Qwen-2.5-Coder^{32B}	**65.00**	**90.00**	**60.00**	**75.00**	**60.00**	**70.00**	

Table 1
Performance of different LLMs on the validation and test sets, 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.

An example of this behavior with an output of Codestral^{22B}:

```
```python
import pandas as pd

def answer(df: pd.DataFrame) -> list:
    column_name = 'considerando una escala de 0 a 10, donde 0 significa '
        nada, en absoluto' y 10 'totalmente', digame, por favor, si durante
        la ultima semana se ha sentido..._Feliz'
    return df[column_name].value_counts().nlargest(3).index.tolist()
```

This function takes a DataFrame `df` as input and returns a list of the
    three most common responses to the question about feeling happy. The `
    value_counts()` function is used to count the occurrences of each
    response, and `nlargest(3)` is used to select the three most common
    responses. The `index` attribute is used to get the actual responses,
    and `tolist()` is used to convert the index to a list.
```

Meanwhile, Qwen-2.5-Coder^{32B} does not make these kind of errors, and just outputs the desired code without need of further postprocessing. It's important to note that the Qwen-2.5-Coder^{32B} model was able to perform better with list datatypes rather than with numbers, when the former datatype tends to stem from more difficult or complex queries.

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 pre-selection. The AG+CS (AG with Column Selector) setup shows a clear improvement of 8 points 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 as the setup with (AG+ECS). It deals better with some datatypes, and worse or equal with others. This means that the model is powerful enough to not input erroneous code, thus the mistakes it makes are from not interpreting the question correctly and giving a wrong answer.

		boolean	category	number	list[category]	list[number]	μ
Valid.	AG	60.00	63.64	**45.45**	66.67	**72.22**	61.00
	AG+CS	**80.00**	86.36	40.91	72.22	66.67	**69.00**
	AG+ECS+CF	65.00	**90.91**	**45.45**	**77.78**	66.67	**69.00**
Test	AG	65.00	**90.00**	60.00	**75.00**	60.00	70.00
	AG+CS	**90.00**	**90.00**	**80.00**	60.00	**70.00**	78.00
	AG+ECS+CF	**90.00**	**90.00**	**80.00**	65.00	**70.00**	**79.00**

Table 2
Performance on the validation and test sets when integrating different components of the pipeline with Qwen-2.5-Coder^{32B} as backbone. The best performance is highlighted in bold.

5.2. Final Test Phase

The best performing configuration is almost a draw between AG+ECS and AG+ECS+CF, but we decided to go with the latter one to participate in the competition, just in case the code fixer is able to correct possible coding mistakes. Our zero-shot approach reached 79 points of accuracy in the task, which ranked us in the 4th position out of 7 participants.

Our results during the development phase (69 points) benefited from a significant increase of 10 points in accuracy with respect to the validation results, likely due to the complexity of the questions present in the test set.

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 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 PRESTA task, 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 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)

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT for minor copy-editing, and GitHub copilot for code autocompletion. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication’s content.

References

[1] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, J. Eisenschlos, TaPas: Weakly Supervised Table Parsing via Pre-training, in: D. Jurafsky, J. Chai, N. Schluter, J. Tetreault (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, 2020, pp. 4320–4333. URL: https://aclanthology.org/2020.acl-main.398/. doi:10.18653/v1/2020.acl-main.398.

[2] P. Yin, G. Neubig, W.-t. Yih, S. Riedel, TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data, in: D. Jurafsky, J. Chai, N. Schluter, J. Tetreault (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, 2020, pp. 8413–8426. URL: https://aclanthology.org/2020.acl-main.745/. doi:10.18653/v1/2020.acl-main.745.

[3] V. Zhong, C. Xiong, R. Socher, Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning, 2017. URL: https://arxiv.org/abs/1709.00103. arXiv:1709.00103.

[4] T. Yu, Z. Li, Z. Zhang, R. Zhang, D. Radev, TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL generation, in: M. Walker, H. Ji, A. Stent (Eds.), Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), Association for Computational Linguistics, New Orleans, Louisiana, 2018, pp. 588–594. URL: https://aclanthology.org/N18-2093/. doi:10.18653/v1/N18-2093.

[5] V. Pal, A. Yates, E. Kanoulas, M. de Rijke, MultiTabQA: Generating Tabular Answers for Multi-Table Question Answering, in: A. Rogers, J. Boyd-Graber, N. Okazaki (Eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Toronto, Canada, 2023, pp. 6322–6334. URL: https://aclanthology.org/2023.acl-long.348/. doi:10.18653/v1/2023.acl-long.348.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language Models are Few-Shot Learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, volume 33, Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[7] Y. Cao, S. Chen, R. Liu, Z. Wang, D. Fried, API-Assisted Code Generation for Question Answering on Varied Table Structures, in: H. Bouamor, J. Pino, K. Bali (Eds.), Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Association for Computational

Linguistics, Singapore, 2023, pp. 14536–14548. URL: https://aclanthology.org/2023.emnlp-main.897/. doi:10.18653/v1/2023.emnlp-main.897.

[8] J. Á. González-Barba, L. Chiruzzo, S. M. Jiménez-Zafra, Overview of IberLEF 2025: Natural Language Processing Challenges for Spanish and other Iberian Languages, in: Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2025), co-located with the 41st Conference of the Spanish Society for Natural Language Processing (SEPLN 2025), CEUR-WS. org, 2025.

[9] J. Osés-Grijalba, L. A. Ureña-López, E. M. Cámara, J. Camacho-Collados, Overview of PRESTA at IberLEF 2025: Question Answering Over Tabular Data In Spanish, in: Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2025), co-located with the 41st Conference of the Spanish Society for Natural Language Processing (SEPLN 2025), CEUR-WS. org, 2025.

[10] P. Rajpurkar, J. Zhang, K. Lopyrev, P. Liang, Squad: 100,000+ Questions for Machine Comprehension of Text, 2016. URL: https://arxiv.org/abs/1606.05250. arXiv:1606.05250.

[11] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, C. D. Manning, HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering, in: E. Riloff, D. Chiang, J. Hockenmaier, J. Tsujii (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Brussels, Belgium, 2018, pp. 2369–2380. URL: https://aclanthology.org/D18-1259/. doi:10.18653/v1/D18-1259.

[12] P. Pasupat, P. Liang, Compositional Semantic Parsing on Semi-Structured Tables, in: C. Zong, M. Strube (Eds.), 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), Association for Computational Linguistics, Beijing, China, 2015, pp. 1470–1480. URL: https://aclanthology.org/P15-1142/. doi:10.3115/v1/P15-1142.

[13] M. Iyyer, W.-t. Yih, M.-W. Chang, Search-based Neural Structured Learning for Sequential Question Answering, in: R. Barzilay, M.-Y. Kan (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Vancouver, Canada, 2017, pp. 1821–1831. URL: https://aclanthology.org/P17-1167/. doi:10.18653/v1/P17-1167.

[14] J. Osés Grijalba, L. A. Ureña-López, E. Martínez Cámara, J. Camacho-Collados, Question Answering over Tabular Data with DataBench: A Large-Scale Empirical Evaluation of LLMs, in: N. Calzolari, M.-Y. Kan, V. Hoste, A. Lenci, S. Sakti, N. Xue (Eds.), Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024), ELRA and ICCL, Torino, Italia, 2024, pp. 13471–13488. URL: https://aclanthology.org/2024.lrec-main.1179/.

[15] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, in: J. Burstein, C. Doran, T. Solorio (Eds.), 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), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186. URL: https://aclanthology.org/N19-1423/. doi:10.18653/v1/N19-1423.

[16] F. Zhou, M. Hu, H. Dong, Z. Cheng, F. Cheng, S. Han, D. Zhang, TaCube: Pre-computing Data Cubes for Answering Numerical-Reasoning Questions over Tabular Data, in: Y. Goldberg, Z. Kozareva, Y. Zhang (Eds.), Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 2022, pp. 2278–2291. URL: https://aclanthology.org/2022.emnlp-main.145/. doi:10.18653/v1/2022.emnlp-main.145.

[17] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, L. Zettlemoyer, BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, in: D. Jurafsky, J. Chai, N. Schluter, J. Tetreault (Eds.), Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, 2020, pp. 7871–7880. URL: https://aclanthology.org/2020.acl-main.703/. doi:10.18653/v1/2020.acl-main.703.

[18] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li,

J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, T. Zhu, Qwen Technical Report, 2023. URL: https://arxiv.org/abs/2309.16609. arXiv:2309.16609.

[19] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7B, 2023. URL: https://arxiv.org/abs/2310.06825. arXiv:2310.06825.

[20] J. O. Grijalba, L. A. U. López, J. Camacho-Collados, E. M. Cámara, Towards quality benchmarking in question answering over tabular data in spanish, Proces. del Leng. Natural 73 (2024) 283–296. URL: http://journal.sepln.org/sepln/ojs/ojs/index.php/pln/article/view/6617.

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.
        Always use the original column names from the dataset, don't change them.
        Don't enclose the code in any special characters or quotes, output only the code.
        Don't use Counter from collections library.
        Don't use any external libraries, only Pandas.
        Don't use pd.to_datetime as the dates in the dataset cannot be converted to datetime.
        Also, don't use pd.unique as it doesn't work with Categorical columns.
        Convert any numpy values to Python values. E.g. np.float64(1.0) to 1.0.


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:
        ''' Retain original column names: df.columns = {list(df.columns)} '''
        # 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}
```