(cc) BY-NC-ND

Automated Code Generation for Tabular Question Answering in Zero-Shot Settings

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

LyS Group, Faculty of Computer Science, Universidade da Coruña, 15071 A Coruña, Spain
Centro de Investigación CITIC, Universidade da Coruña, 15071 A Coruña, Spain
Correspondence: adrian.lopez.gude@udc.es, roi.santos.rios@udc.es

DOI: *https://doi.org/10.17979/spu.23.c60*

Abstract: Our work presents an effective and robust approach to Tabular Question Answering based on a zero-shot pipeline that uses a Large Language Model to generate executable code for extracting information from tables. Our system includes a modular architecture: a code generation module supported by column selection and type inference components to guide accurate code synthesis, together with an iterative repair mechanism for runtime errors into revised prompts. Our approach is successfully tested both in English and Spanish.

1 Introduction

Tabular Question Answering (Tabular QA) has huge potential in real-world applications 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, 2020; Yin, 2020) or by formulating the task as a sequence-to-sequence problem (Pal, 2023; Yu, 2018; Zhong, 2017). However, with the rise of instruction-based Large Language Models (LLM), recent approaches have shifted away from reliance on large annotated datasets, instead reframing the task as a zero-shot generation problem (Cao, 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.[1]

2 Background

Early tasks in Question Answering (QA) mainly focused on retrieving information from unstructured text sources (Ojokoh, 2018), but the increasing availability of structured datasets

[1] Our implementations are fully available at *https://github.com/adrian-gude/Tabular_QA* for English and *https://github.com/Dunque/Tabular_QA_es* for Spanish.

has led to new challenges in understanding and querying tabular data. For this purpose, 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.

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, 2020) feeds both the input question and the flattened table into BERT (Devlin, 2019) as a single sequence, and finetunes the architecture to select relevant columns and predict an aggregation function. Similarly, TACUBE (Zhou, 2022) combines a cube constructor with BART (Lewis, 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 the use of generative models for program synthesis, where an LLM is finetuned to generate executable programs or instructions (e.g., as SQL queries) to be applied on tabular sources. Zhong (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 (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, 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, 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 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, 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 to answer the query.[2]

[2] Our prompts are both available in the code, publicly available at GitHub.

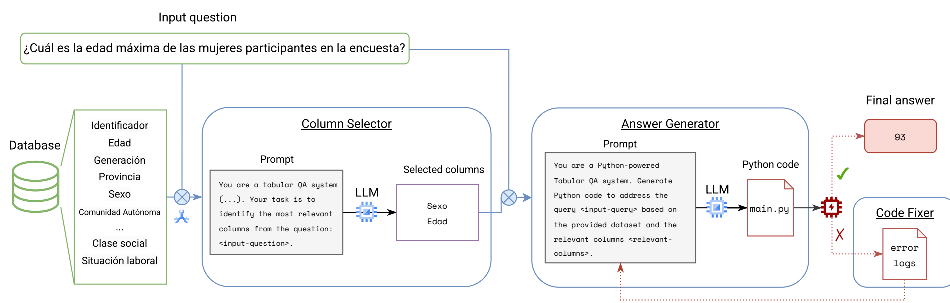

Figure 1: Architecture of our system, with an example query in Spanish. Different symbols are used to represent different elements of our pipeline: ⊗ merges information in a prompting-like form, represents a preprocessing step, indicates LLM inference (with optional post-processing steps), and runs Python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

		bool.	cat.	num.	list[cat.]	list[num.]	μ	β
English	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**	
Spanish	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**	

Table 1: Performance of different LLMs on the validation set for the English and Spanish datasets, 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.

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 Our hardware resources are somewhat limited by today's standards: shared access to an Intel Core i9-10920X at 3.50 GHz with 258 GiB RAM and two integrated NVIDIA RTX 3090. We conducted experiments with different open-source LLMs adjusted to these hardware limitations, specifically pretrained for instruction-based code generation: Qwen-2.5-Coder (both 7B and 32B versions) (Bai, 2023), 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 with the datasets. 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 for each language.

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 Spanish baseline system is Qwen-2.5-Coder^{32B}, while all LLMs outperform the English baseline. Still, the best results are obtained with Qwen-2.5-Coder^{32B} in both languages.

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 Spanish dataset. This explains the subpar results of these models.

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. Next, we show an example of this behavior with an output of Codestral^{22B} for the Spanish dataset:

```
'''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.

Regarding the English dataset, the models tend to perform better with the simpler datatypes, and struggle more with the list types.

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). As previously shown, the best model for both languages was Qwen-2.5-Coder^{32B}. Table 2 shows the results when varying the components of the pipeline while maintaining this LLM 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 in both languages 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) in Spanish, dealing 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. Meanwhile, in English the results are again improved regarding all datatypes.

		boolean	category	number	list[category]	list[number]	μ
English	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**
Spanish	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**

Table 2: Performance on the validation set for both languages 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

In Spanish, the best performing configuration is almost a draw between AG+ECS and AG+ECS+CF, but in English there is a clear advantage when using the latter configuration. Thus, we decided to use it for both languages. In Spanish, our system achieved 79 points of accuracy in the task. Compared to 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.

In English, our system achieved 74.71 points of accuracy, almost a 11 point drop compared to the results obtained with the validation set. Again, this is attributed to the complexity of the expected datatypes, as most of the questions required to parse through non-standard list formats in the tables.

6 Conclusions and Future Work

In this work we propose a zero-shot approach for Tabular QA both that demonstrated a strong performance in English and Spanish. 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.

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). 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 (ED431G 2023/01).

Bibliography

J. e. a. Bai. Qwen Technical Report, 2023. URL *https://arxiv.org/abs/2309.16609*.

Y. e. a. Cao. API-Assisted Code Generation for Question Answering on Varied Table Structures. In H. e. a. Bouamor, editor, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14536–14548, Singapore, Dec. 2023. ACL. URL *https://aclanthology.org/2023.emnlp-main.897/*.

J. e. a. Devlin. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In J. e. a. Burstein, editor, *Proceedings of the 2019 NAACL Conference: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. ACL. URL *https://aclanthology.org/N19-1423/*.

J. e. a. Herzig. TaPas: Weakly Supervised Table Parsing via Pre-training. In D. e. a. Jurafsky, editor, *Proceedings of the 58th Annual Meeting of the ACL*, pages 4320–4333, Online, July 2020. ACL. URL *https://aclanthology.org/2020.acl-main.398/*.

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, and W. E. Sayed. Mistral 7B, 2023. URL *https://arxiv.org/abs/2310.06825*.

M. e. a. Lewis. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In D. e. a. Jurafsky, editor, *Proceedings of the 58th Annual Meeting of the ACL*, pages 7871–7880, Online, July 2020. ACL. URL *https://aclanthology.org/2020.acl-main.703/*.

B. e. a. Ojokoh. A review of question answering systems. *Journal of Web Engineering*, 17(8): 717–758, 2018.

V. e. a. Pal. MultiTabQA: Generating Tabular Answers for Multi-Table Question Answering. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the ACL (Volume 1: Long Papers)*, pages 6322–6334, Toronto, Canada, July 2023. ACL. URL *https://aclanthology.org/2023.acl-long.348/*.

P. e. a. Yin. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In D. e. a. Jurafsky, editor, *Proceedings of the 58th Annual Meeting of the ACL*, pages 8413–8426, Online, July 2020. ACL. URL *https://aclanthology.org/2020.acl-main.745/*.

T. e. a. Yu. TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL generation. In M. e. a. Walker, editor, *Proceedings of the 2018 NAACL Conference: Human Language Technologies, Volume 2 (Short Papers)*, pages 588–594, New Orleans, Louisiana, June 2018. ACL. URL *https://aclanthology.org/N18-2093/*.

V. e. a. Zhong. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning, 2017. URL *https://arxiv.org/abs/1709.00103*.

F. e. a. Zhou. TaCube: Pre-computing Data Cubes for Answering Numerical-Reasoning Questions over Tabular Data. In Y. e. a. Goldberg, editor, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2278–2291, Abu Dhabi, United Arab Emirates, Dec. 2022. ACL. URL *https://aclanthology.org/2022.emnlp-main.145/*.