MRT at IberLEF-2025 PRESTA Task: Maximizing Recovery from Tables with Multiple Steps

Maximiliano Hormazábal Lagos[1,*,†], Álvaro Bueno Sáez[1,*,†], Héctor Cerezo-Costas[1,*,†], Pedro Alonso Doval[1,*,†] and Jorge Alcalde Vesteiro[1,*,†]

[1]*Fundación Centro Tecnolóxico de Telecomunicacións de Galicia (GRADIANT), Vigo, Spain*

Abstract

This paper presents our approach for the IberLEF 2025 Task PRESTA: Preguntas y Respuestas sobre Tablas en Español (Questions and Answers about Tables in Spanish). Our solution obtains answers to the questions by implementing Python code generation with LLMs that is used to filter and process the table. This solution evolves from the MRT implementation for the Semeval 2025 related task. The process consists of multiple steps: analyzing and understanding the content of the table, selecting the useful columns, generating instructions in natural language, translating these instructions to code, running it, and handling potential errors or exceptions. These steps use open-source LLMs and fine-grained optimized prompts for each step. With this approach, we achieved an accuracy score of 85% in the task.

Keywords

Table Question Answering, Large Language Models, Code generation

1. Introduction

Natural Language Processing (NLP) is nowadays constrained by the amount of information that can be processed by Large Language Models (LLMs) due to the limited capacity of input data that they can handle. Some applications that have this limitation are response generation using RAG systems in which the LLM generates answers with a small subset of retrieved documents[1] as context. In table questions answering this limitation is aggravated as tables and databases can have millions of records and columns [2], which by today standards will not fit completely in the LLM context.

In this paper, we improve the algorithm of Maximizing Recovery from Tables with Multiple Steps (MRT)[3], a multi-step process that implements LLMs and Python code generation to answer questions as objectively as possible. Our system implements a sequential divide-and-conquer approach in which LLMs or heuristic algorithms are executed at each step with very specific tasks. These steps range from describing the tables and generating instructions in natural language to producing the source code to implement the previous instructions, executing it, and parsing the output to obtain the final answer. In comparison with end-to-end solutions, our approach is more explainable as it is easy to debug and trace which was the cause of a good or bad response.

This paper addresses the IberLEF 2025 [4] Task PRESTA: Preguntas y Respuestas sobre Tablas en Español (Questions and Answers about Tables in Spanish) [5]. The code that generated these results is publicly available[1].

IberLEF 2025, September 2025, Zaragoza, Spain

*Corresponding author.

†These authors contributed equally.

mhormazabal@gradiant.org (M. H. Lagos); abueno@gradiant.org (B. Sáez); hcerezo@gradiant.org (H. Cerezo-Costas); palonso@gradiant.org (P. A. Doval); jalcalde@gradiant.org (J. A. Vesteiro)

0009-0003-3687-1924 (M. H. Lagos); 0009-0006-9199-972X (B. Sáez); 0000-0003-2813-2462 (H. Cerezo-Costas); 0009-0000-8255-3466 (P. A. Doval); 0009-0003-6418-3694 (J. A. Vesteiro)

[1]https://github.com/Gradiant/MRT_TableQA/releases/tag/v2.0.0

2. Background

As the name implies, question answering (QA) consists of answering questions that normally have objectively correct answers. Tabular QA requires the system to retrieve responses from knowledge bases represented as datasets in tables. Recent methods for QA in tabular data, such as TAPAS[6], TAPEX [7] or Omnitab[8], integrate transformers with architectures specifically adapted to extract answers directly from tables as they integrate said tables as context. Parallel to this, LLMs have also been employed in zero-shot and few-shot strategies, and have proven to show high levels of usability in QA due to their prior knowledge. One of the benefits of zero/few shot strategies is the fact that domain-specific fine-tuning could be avoided, eliminating the usual needs of gathering, cleaning, and performing human validation for the task. In addition, recent LLMs include reasoning skills by default. This helps, but still presents difficulties with complex queries, involving multiple columns, large tables, or ambiguous questions requiring common-sense knowledge.

Another approach is to parse natural language questions into formal queries in programming languages such as SQL. There are systems like Seq2SQL[9] or TableGPT2[10], which are designed to create SQL queries from relational database queries or Python code, respectively. These methods are theoretically independent of table size (there is no context limitation) and provide greater transparency by including intermediate steps to supervise generated queries.

Several datasets to evaluate TableQA strategies have been released in the past years. WikiSQL [11] based on Wikipedia and TabFact [12] provided structured evaluation of tabular data. However, these datasets do not convey the heterogeneity of real-world tabular data, which are usually more complex, less homogeneous, and unstructured. [13]. To address this problem, DataBench[14] has been developed, which brings together 65 real-world datasets with more than $1,300$ manually crafted question-answer pairs in multiple domains.

Apart from the methods with LLMs and code generators mentioned above, others use alternative strategies such as Retrieval Augmented Generation (RAG) or Chain-of-Thoughts (CoT). For example, TableRag [15] proposes the use of RAG systems for tabular comprehension tasks, such as QA, employing techniques such as query expansion and a double transformation to query languages. This process translates, on the one hand, the schema to be interacted with and, on the other hand, the operation necessary to identify the cells with the answer. A noteworthy proposal is Chain-of-Table [16], which implements CoT as an iterative reasoning mechanism. Instead of executing the code in one shot, programming instructions are executed iteratively to add or discard information from the table until the final answer is found.

Although there has been recent progress in this field, certain challenges still persist, such as the enhancement of reasoning across multiple rows and columns, managing multiple domains and languages, mixing data from multiple tables, or improving explainability.

3. Background of this task

Similar tasks to PRESTA challenge have been recently published. This is the case of SemEval 2025 Task 8: Question-Answering over Tabular Data challenge [17], a task with the same purpose as this, mainly in English, with tables from other contexts and a wider range of topics.

For this task, we developed MRT: Maximizing Recovery from Tables with Multiple Steps [3]. This solution treats the tables as Pandas Dataframes and uses LLMs to generate Python code that could extract an answer to each question. We now present briefly the MRT system, which is our starting point in the PRESTA challenge.

The initial step is the *column descriptor* module, which analyses the data of each column (description of the column content meaning, type of data, frequent values, max and min values, etc.). Secondly, the *explainer* module, using the previous analysis, generates natural language instructions with an LLM. These instructions contain the steps needed to get the answer. Then, the *coder* module generates, also with an LLM, Python code from the text instructions, and afterward, this code is executed by the *runner*

module. If an exception occurs during the code execution or answer parsing, the system steps back into the *coder* in an iterative looping process until it gets a valid answer or a limit of attempts is exceeded. Finally, the *interpreter* module and *formatter* module implement different approaches for obtaining the answer in the desirable data type in order to match the expected result for the task. In each step, the same or different LLM could be used. Usually, we employ an LLM finetuned for code generation within the *coder* module.

Some of the limitations of MRT were:

- It struggles to filter categorical values when the value in the dataset has a different representation as it appears in the question. For example, when asked for *Obama* the system may not find the value with a strict match if the representation in the table is *Barack Obama* instead.
- The generated natural language instructions were more intricate than they need to be to get the right response. Sometimes filtering instructions were added that are not needed to obtain the response. The generated code produced several exceptions with filters that at first glance are actually easy to execute.
- It does not scale well with a large number of columns. Some modules include in the prompt all the columns of the table, their descriptions and statistical information and frequent values. However, this does not scale properly when the number of columns is large.

4. System overview

The system with the different components is presented in Figure 1. New components were added from the previous version, *column selector* and others were deeply modified (*explainer*, *coder* to make the system more resilient to the new challenges posed by the PRESTA challenge.

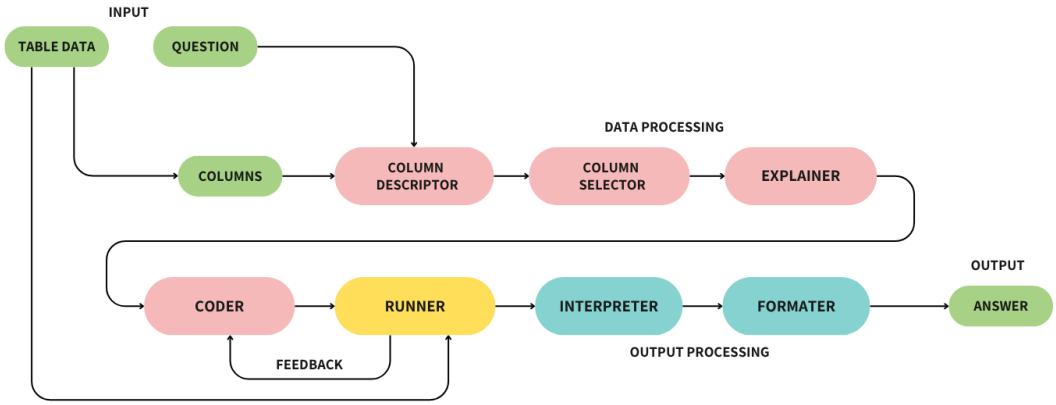

Figure 1: Diagram of the system showing all the steps involved in the generation of the response.

4.1. New challenges addressed in this task

Compared with the SemEval task, the PRESTA dataset patented new limitations that our former MRT system had:

- A larger number of columns in each table implied enormous prompts in the *explainer* module. Sometimes, this leads to exceptions running LLMs. Having more columns increases the changes of selecting wrong columns to answer a question. For comparison, Semeval task tables had an average of 24.8 columns, while in this IberLEF task they have an average of 174.1 columns.

- Ambiguous names. Some columns have names that without all the context (or even with it) are not informative about the content of their cells, or that use initials that are not intuitive. In those cases, their values are difficult to interpret even by humans. Some clarifying examples are *N_R*, *¿Usted es? (LEER_SÓLO PARA LOS QUE HAN CONTESTADO QUE "TRABAJA" EN LA P2014)* or other boolean columns that are possible answers to question, which could not be easily inferred by the name of the column: *Analgésico_antiinflamatorio_1*, *Antidepresivos (fluoxetina, sertralina, escitalopram)_1*...
- Columns with mixed types. For example, columns that are essentially numerical but actually contain strings for some of the values. This is the case of columns like *Del 1 al 10, con qué probabilidad votarías al partido político BNG?* (From 1 to 10, with which probability you would vote to BNG party?), whose values are 5, 6, 7, etc., but also contain these options *1 - No le votaría nunca*, *10 - Le votaría siempre*. Using numerical operations directly in these columns would throw exceptions.

4.2. Development of new features

To mitigate and correct both the already known limitations inherited from the original MRT work and face the new challenges found in the training dataset for PRESTA, we developed and improved some features.

4.2.1. Column selector module

We designed the *column selector* module to make an initial filter of columns before feeding the prompt of the *explainer*, to avoid huge prompts that can lead to exceptions or errors. Due to its nature in the current implementation, the *explainer* has to know in advance all the columns involved in answering the question and their descriptions in order to obtain correct natural language instructions. Those questions must be included in the prompt with the question. Henceforth, this new module tries to filter the columns to avoid crashes by large prompts whilst leaving the relevant questions untouched.

This new step uses an LLM to ask which columns are potentially relevant to that question. To achieve this, it prompts iteratively the LLM in groups of 25 columns each, giving their names and descriptions along the question. The prompt emphasizes that, in case of doubt, it should return the column, in order to not miss the relevant information in this step.

The output of this module is the set of columns that the LLM considered useful for the *explainer* module, conditioned by the query.

4.2.2. Rules for removing uninformative columns

Column names can be ambiguous and mislead the different modules. We identified exceptional columns that led to significant and recurrent errors, that we consider are not needed in any of the questions seen and will not make sense to use in questions of this style. This is the case of *"N_R_" ("No Recuerda"/"Don't remember")*, which the *column descriptor* wrongly described as "Number of respondents". Having bad descriptions will produce wrong answers if those columns are used to obtain the answer instead of the good ones.

Columns consisting of numerations or that share almost the name with others, where it is difficult to identify semantic differences between them, were discarded. For example, there is one table that has columns identified as *"Ns_Nc_0"*, *"Ns_Nc_1"*, *"Ns_Nc_2"*, etc.

4.2.3. Clarification instructions in natural language explanations

The original MRT code returned raw text instructions that were then parsed to obtain a list of steps that should be coded afterwards.

The evolution over the original MRT implementation corrects mistakes due to the wrong naming of columns or variables in the natural language instructions.

In the new version, the raw text format using in the output was changed to a JSON with the following properties: *instructions*: the list of natural language instructions that must be applied to the data to get the answer, *columns*: the list of columns used in the instructions and *filter_values*, values that will be used to filter the information of the table.

After the natural language instructions are calculated a check is performed to correct the column names that are misspelled by the model. Levenshtein distance is used to obtain the closest column to the one appearing in the instructions, directly switching the name if they are not equal.

With the *filter_values* a similar procedure was implemented. Instead of substituting the values, a clarification instruction was added to those originally generated by the system including the old value. Those instructions have the following format *"Be careful!. The value ⟨old value⟩ appears in the database with the following format: ⟨new value⟩"*.

Furthermore to simplify the task for the *coder* information about the column types and the typical values of the column (only for the not-numerical columns) are added at the end of the instructions: *"The column ⟨column name⟩ is of type ⟨column type⟩ and has the following example values: ⟨column values⟩"*.

Table 1 shows an example without clarification instructions and with clarification instructions. The instructions on the right are more complete and easier to understand than their counterparts on the left that lack certain information and are more prone to fail.

W/o clarification Instructions	**With clarification Instructions**
1) Count the total number of surveys conducted in January 2) Compare the count of surveys conducted in January with the total count of surveys to determine if most surveys were conducted in January.	1) Count the total number of surveys conducted in January 2) Compare the count of surveys conducted in January with the total count of surveys to determine if most surveys were conducted in January. 3) Be careful! The value enero appears in the database with the following format: 'Enero' 4) The column 'Mes de realización' is of type 'object' and has the following example values: Enero, Febrero, Marzo

Table 1
Natural language instructions generated by the *explainer* with and without clarification instructions for the question `¿Fueron la mayoría de las encuestas realizadas en enero?`

4.2.4. Custom Functions for Code Generation

Assigning the full responsibility of both orchestrating and generating code to a single model could overload its capabilities. Empirical results supported this hypothesis, as evidenced by recurrent instances of poor coding practices despite explicit instructions to avoid them. Common problematic patterns included:

1. Misuse of the `group_by` function, leading to code errors instead of employing suitable alternatives that are less prone to produce exceptions.
2. Contradictory ordering operations, such as sorting in descending order initially, followed by an explicit ascending sort, effectively negating previous instructions.

Inspired by the methodology of [18], though without precomputing function-result cubes, we developed instead pre-coded generic functions that can be used in multiple contexts but that solve many of these common mistakes. The prompt conditions the model to use these functions as an alternative to the *pandas* implementations. This allowed the model to primarily focus on code orchestration, significantly reducing its code-generation workload.

The objective behind these functions was to maintain general applicability rather than targeting overly specific scenarios. For instance, functions are generalized to handle common data tasks, such as counting occurrences or filtering data based on numerical conditions.

These functions were created through a semi-automated approach involving both Large Language Models (LLMs) and human input:

1. An LLM analyzed the training dataset and proposed generic function templates that could broadly address the questions of the dataset. Additionally, the LLM had the option to reuse existing templates. The outcome was a set of function templates capable of solving a substantial portion of the queries. This functions were obtained from the training split.
2. Human developers implemented these function templates, subsequently verifying that the model actively employed them, therefore improving empirical performance.
3. During the testing and debugging phase, we validated that pre-coded, generalized functions effectively resolved recurring issues, prompting further expansion of the function pool.
4. Additionally, certain functions incorporated fuzzy decision-making capabilities to enhance performance, a methodology detailed in the subsequent section, *Fuzzy Search of Categorical Values*.

The final set of developed generic functions is in appendix I.

Additionally, we specifically addressed the challenge of columns misidentified as numerical due to naming conventions through the dedicated internal function `extract_numeric`, detailed in the Appendix A.

Lastly, another way to conceptualize this methodology is by viewing the provision of these functions as enabling the model to utilize Python tool-calling capabilities, directly resolving common coding problems encountered by the model [19].

4.2.5. Fuzzy Search of Categorical Values

As previously described, certain functions within the model pipeline employ fuzzy matching techniques to mitigate typical errors encountered during data processing, particularly those stemming from inconsistencies or typographical variations in categorical values. These errors frequently arise due to the complexity and variability of data names, especially after undergoing multiple processing stages.

To clarify the motivation behind implementing fuzzy matching, we first outline the challenges experienced in earlier iterations of the pipeline and also in the version presented in this paper. Specifically, accessing column and row names in a reliable manner posed substantial difficulties due to discrepancies that emerged across the five stages of large language model (LLM) processing. Such discrepancies typically result in false positives and false negatives:

1. **False positives** occur when the model incorrectly forwards values due to minor deviations in the text, such as pluralization or capitalization inconsistencies. For example, the value "item" might erroneously be transformed into "items" or "Item."
2. **False negatives** occur when originally correct but unusually formatted values are improperly corrected, thereby introducing errors. For instance, the value "iTem" might incorrectly be normalized to "item," altering the intended representation.

To address these challenges, we adopt established fuzzy matching methodologies as detailed in prior research such as [20]. Rather than relying solely on exact matches—which fail to accommodate minor textual variations—fuzzy matching techniques allow the model to recognize and utilize values that closely resemble the target values based on a defined similarity threshold.

An illustrative example of such a fuzzy matching approach in our pipeline is presented through the Python functions shown in appendix B.

These functions perform sequentially the following steps:

1. Identifying the target column and value.

2. Recognize empirically that exact matches are often not achievable due to textual variations.
3. If an exact match is unavailable, apply fuzzy matching in a secondary step, selecting the closest matching value based on the predefined similarity threshold and subsequently operating on it.

In practice, this fuzzy matching strategy enhances significantly the robustness and overall accuracy of the pipeline, effectively reducing the incidence of errors due to minor textual discrepancies.

5. Experimental setup

The experimental setup was executed in batches, where all the questions in the dataset were run through one step before advancing to the following one. Thus, the number of times a model has to be loaded/unloaded was optimized as each of the steps may use different models. Also, results of the column descriptor were cached between experiments and was executed only once, given that its output for a table is independent of the questions.

The tests were executed in a NVIDIA RTX-a6000 that combines 84 second-generation RT cores, 336 third-generation Tensor cores, and 10,752 CUDA cores with 48 GB of graphics memory for performance.

5.1. Dataset splits

Although no training of any model has been performed, the splits of the dataset are shown below (see table 2). The tables used in the test consist of the same tables of the train and the dev splits.

Split	Tables	Questions
train	6	150
dev	4	100
test	10	100

Table 2
Distribution of number of tables and questions for each split in the dataset

Train and dev splits have been used for the development of the modules, whereas test split was solely used for the validation of the system against the official platform used in the benchmark.

5.2. Models

We decided to use Qwen[2] models for the different modules of the system. Specifically, we executed two types of Qwen models: Qwen 2.5 14B[3] for all the modules, excepting the coder which used Qwen 2.5 coder 14B[4].

We also made some tests using the recently published model Qwen 3 [5] in the explainer, maintaining Qwen 2.5 and Qwen 2.5 coder for the rest of the modules.

6. Results

6.1. Performance in validation test

Table 3 shows the results obtained in validation data in our main scenario (executing with Qwen 2.5 and Qwen 2.5 coder). We can see the performance for each type of data. The total score is 71%, while for most of the data types are very homogeneous, varying from 75% to 80%, except for numerical answers, which clearly perform worse with a 50% score.

[2]https://huggingface.co/Qwen
[3]https://huggingface.co/Qwen/Qwen2.5-14B-Instruct
[4]https://huggingface.co/Qwen/Qwen2.5-Coder-14B-Instruct
[5]https://huggingface.co/Qwen/Qwen3-14B

	Total	Boolean	Number	Category	List[Category]	List[Number]
Score	0.71	0.75	0.5	0.77	0.8	0.75
Size	100	20	22	22	20	16

Table 3
Score and number of answers per answer type in the validation split.

6.2. Performance in test set

We submitted 3 results to the task. One is with the results of our system using Qwen 2.5 14B for all the steps that involve LLMs, except the coder which used Qwen Coder 14B. The second one selects the output of the interpreter module in that same execution. Finally, a third one changes only the model of the explainer to Qwen 3 14B. They achieve **scores of 85%, 85%, and 83%** respectively. We show in the table 4 our best submission broken down by type of expected answer. All the experiments were repeated 8 times, taking the most repeated answer (a simple majority voting strategy) as the final result.

	Total	Boolean	Number	Category	List[Category]	List[Number]
Score	0.85	0.95	0.9	0.8	0.8	0.75
Size	100	20	20	20	20	20

Table 4
Score and number of answer per answer type in the test split.

In comparison with the score of validation set, in this case is 15% higher. It matches our intuition that is that the test set questions are in average simpler than the validation set questions.

6.3. Manual Error Analysis in the validation set

We performed a manual error analysis of the answers for the validation set flagged as an error by the evaluator. The results are summarized in Table 5. The main source of errors is the wrong generation of instructions in the *explainer* module. Some of the errors involve not selecting the correct column to use, although sometimes the ambiguity of the column names makes this choice difficult. For example, columns such us 'Edad' vs 'Edad_recodificada' in which one is just a higher level of abstraction from the other. Other cases involved just wrong natural language instructions. Adding unnecessary extra filters (removing nulls, zeros, empty lists, etc.) was very frequent source of errors.

The other main source of errors is the removal of relevant columns in the column selector. Sometimes it struggles with columns that have long names and even involve complex semantics such as conditionals, like survey questions present in this dataset.

We identified a few errors in the interpreter for cases where the runner actually obtained the correct response. This case usually involves incorrectly removing symbols or clarifications. For instance, there are two related to age intervals: "+65" and "18-24" which were changed to "65" and "1824". Other less frequent errors were due to the transformations of the output that make the metric implemented to consider it as an error. The removal of the parenthesis and the information within in 'PP' in the expected answer 'PP (Partido Popular)'. This last example is flagged as incorrect according to the benchmark metrics but that could be perfectly be deemed as acceptable by common sense with human feedback.

Compared to manual error analysis in our previous work for the analogous SemEval task, we can prove the benefits of some of our new features taking into account that wrong cell filtering is not an issue anymore as it was before and code errors and code exceptions have been notably reduced.

6.4. Ablation study

We have performed an ablation study to evaluate the impact of some of the features on the performance of our system. To do this, we deactivate one by one the new modules and execute the benchmark with the validation set. Every configuration is repeated 8 times, taking a majority voting ensemble as the

Description	Errors	% error
Wrong Instructions	11	37.9%
Wrong column filtering	9	31.0%
Formatting (transformations)	3	10.3%
Code Generation (incl. exceptions)	2	6.6%
Others	4	13.7%

Table 5
Manual analysis of the errors in the validation split

final result, discarding thrown exceptions or bad results (e.g. responses such as *No matching records were found*).

Table 6 shows the scores of each of the tries broken down by the type of expected answer. As can be seen, the overall metric is always in the range of 0.69 and 0.74. The number of question-answer pairs is very low (100) and the diversity of tables used in the validation set (4) is not enough to confirm whether the new modules improved or not the performance of the system in the test. For example, the *column selector* module was implemented to avoid throwing exceptions when the number of columns was very large. Nevertheless, none of the 4 datasets used in the validation throws this exception. Hence filtering the columns could have a detrimental effect if a good column was removed. Filtering out columns has a positive impact on time consumption, as the execution of the overall system is much faster (e.g. about three times in our experiments within the PRESTA dataset).

Scenario Scenario	Score in dev	Boolean	Number	Category	List Category	List Number
All (formatter)	0.71	0.75	0.5	0.77	0.8	0.75
All (Interpreter)	0.71	0.75	0.5	0.77	0.8	0.75
w/o column selector	0.74	0.8	0.55	0.82	0.8	0.75
w/o custom functions	0.72	0.75	0.55	0.77	0.75	0.81
w/o explainer corrector	0.7	0.7	0.5	0.77	0.8	0.75
w/o retries in coder	0.71	0.7	0.5	0.77	0.85	0.75
w/o fuzzy subs.	0.69	0.8	0.5	0.73	0.8	0.63

Table 6
Score in each of the scenarios in the validation set of the ablation study broke down by type of data.

By seeing these results, one question that might arise is whether all the experiments are failing in the same questions. In other words, if the errors in the validation set are not addressed by the new modules. Figure 2 shows the error repetition frequency of errors in all the experiments. We can see that more than half of the errors are coincident in all the experiments.

Combining the information of the most repeated errors (6 and 7 times) with the information of the manual analysis, we find that most errors came from the bad selection of columns (9) or the *explainer* not being able to generate good instructions in natural language (7). Some errors were formatting issues (2) and errors due to the validation metric but they were essentially correct (3).

Finally, we can observe the effect of using an ensemble with majority voting in Figure 3. An ensemble of 5 experiments should be enough, although we have used 8 repetitions in our configuration.

7. Conclusions

In this work, we presented our system MRT for answering queries over tables. Our system generates Python code involving multiple steps: describing and filtering columns, generating natural language instructions, code generation, and formatting the answer. This strategy builds upon our previous work in which we addressed some of the common sources of problems made by the previous version: adding auxiliary functions to avoid recurring exceptions, fuzzy match of cell and column names to improve the understanding of the question and instructions, and the selection of relevant columns to avoid

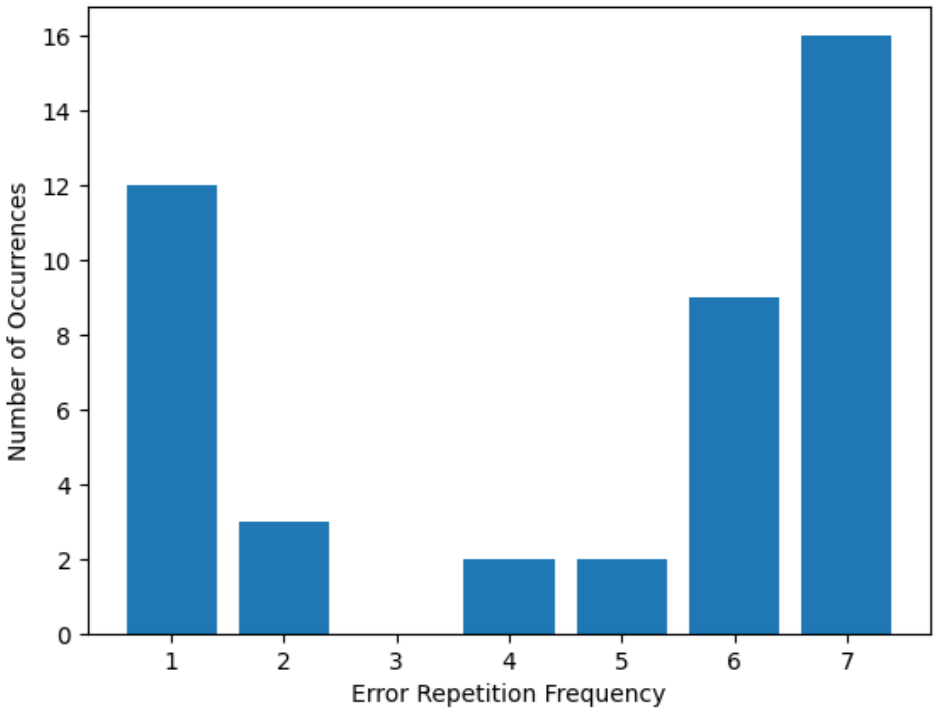

Figure 2: Error repetition frequency in all the 7 experiments in the validation set

LLM exceptions due to context size and improving the speed of the system at the same time. We used middle-size pretrained LMs with $14B$ parameters achieving a third place in the task with a 0.85% of accuracy. One clear benefit of our approach is its explainability as the user can very easily understand what is the source of the errors by seeing the natural language instructions and the generated code.

However, evaluating the benefits of the theoretical improvements is difficult as the dataset lacks the size and diversity in order to be statistically relevant. The differences between the configurations are very small (between 1 and 5 question/answer pairs). In the future, we plan to test the system against larger datasets in order to gain more insights into the relevance of each block in the final answer.

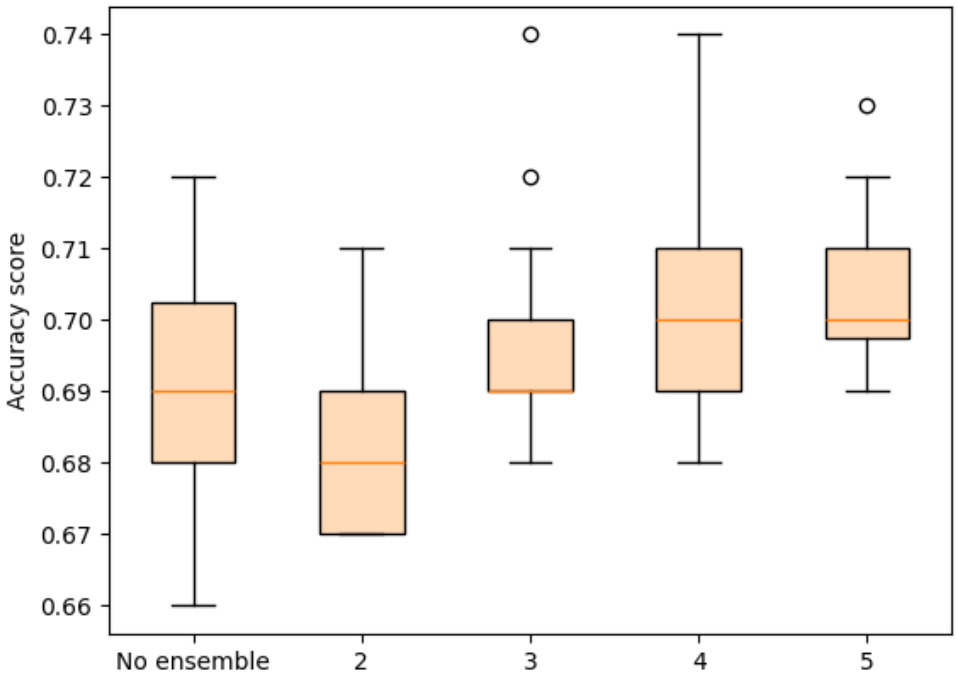

Figure 3: Accuracy score using an ensemble with majority voting

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly in order to: Grammar and spelling check and GPT-4 in order to: suggestions for academic writing style. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] T. Liu, F. Wang, M. Chen, Rethinking Tabular Data Understanding with Large Language Models, in: K. Duh, H. Gomez, S. Bethard (Eds.), Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), Association for Computational Linguistics, Mexico City, Mexico, 2024, pp. 450–482. URL: https://aclanthology.org/2024.naacl-long.26/. doi:10.18653/v1/2024.naacl-long.26.

[2] Y. Ruan, X. Lan, J. Ma, Y. Dong, K. He, M. Feng, Language modeling on tabular data: A survey of foundations, techniques and evolution, 2024. URL: https://arxiv.org/abs/2408.10548. arXiv:2408.10548.

[3] M. Hormazabal-Lagos, Álvaro Bueno Saez, H. Cerezo-Costas, P. A. Doval, J. A. Vesteiro, MRT at SemEval-2025 Task 8: Maximizing Recovery from Tables with Multiple Steps, in: Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025), Association for Computational Linguistics, Vienna, Austria, 2025.

[4] J. 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.

[5] 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.

[6] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, J. Eisenschlos, TaPas: Weakly Supervised Table Parsing via Pre-training, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 2020. URL: http://dx.doi.org/10.18653/v1/2020.acl-main.398. doi:10.18653/v1/2020.acl-main.398.

[7] Q. Liu, B. Chen, J. Guo, M. Ziyadi, Z. Lin, W. Chen, J.-G. Lou, Tapex: Table Pre-Training via Learning a Neural SQL Executor, arXiv preprint arXiv:2107.07653 (2021).

[8] Z. Jiang, Y. Mao, P. He, G. Neubig, W. Chen, OmniTab: Pretraining with Natural and Synthetic Data for Few-Shot Table-based Question Answering, arXiv preprint arXiv:2207.03637 (2022).

[9] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, Y. Luo, A Survey of NL2SQL with Large Language Models: Where are we, and Where are we Going?, arXiv preprint arXiv:2408.05109 (2024).

[10] A. Su, A. Wang, C. Ye, C. Zhou, G. Zhang, G. Chen, G. Zhu, H. Wang, H. Xu, H. Chen, et al., TableGPT2: A Large Multimodal Model with Tabular Data Integration, arXiv preprint arXiv:2411.02059 (2024).

[11] 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.

[12] W. Chen, H. Wang, J. Chen, Y. Zhang, H. Wang, S. Li, X. Zhou, W. Y. Wang, TabFact: A Large-scale Dataset for Table-based Fact Verification, 2020. URL: https://arxiv.org/abs/1909.02164. arXiv:1909.02164.

[13] W. Hwang, J. Yim, S. Park, M. Seo, A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization, 2019. URL: https://arxiv.org/abs/1902.01069. arXiv:1902.01069.

[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] S.-A. Chen, L. Miculicich, J. M. Eisenschlos, Z. Wang, Z. Wang, Y. Chen, Y. Fujii, H.-T. Lin, C.-Y. Lee, T. Pfister, TableRAG: Million-Token Table Understanding with Language Models, 2024. URL: https://arxiv.org/abs/2410.04739. arXiv:2410.04739.

[16] Z. Wang, H. Zhang, C.-L. Li, J. M. Eisenschlos, V. Perot, Z. Wang, L. Miculicich, Y. Fujii, J. Shang, C.-Y. Lee, T. Pfister, Chain-of-Table: Evolving Tables in the Reasoning Chain for Table Understanding, 2024. URL: https://arxiv.org/abs/2401.04398. arXiv:2401.04398.

[17] J. Osés Grijalba, L. A. Ureña-López, E. Martínez Cámara, J. Camacho-Collados, SemEval-2025 Task 8: Question Answering over Tabular Data, in: Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025), Association for Computational Linguistics, Vienna, Austria, 2025.

[18] F. Zhou, M. Hu, H. Dong, Z. Cheng, S. Han, D. Zhang, Tacube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data, 2022. URL: https://arxiv.org/abs/2205.12682. arXiv:2205.12682.

[19] S. He, Achieving tool calling functionality in llms using only prompt engineering without fine-tuning, 2024. URL: https://arxiv.org/abs/2407.04997. arXiv:2407.04997.

[20] S. Sheu, A. Chang, W. Huang, Fast similarity search in string databases, in: 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers), volume 1, 2005, pp. 617–622 vol.1. doi:10.1109/AINA.2005.185.

A. Appendix I: Custom functions for coder

This section contains the definitions of the functions that have been implemented to help the coder to perform some common operations.

```
def flatten_column_values_from_df(df: pd.DataFrame, column: str) -> pd.DataFrame:

def get_top_n_records_with_non_nan_column_value(
df: pd.DataFrame, column: str, number: int
) -> pd.DataFrame:

def get_tail_n_records_with_non_nan_column_value(
df: pd.DataFrame, column: str, number: int
) -> pd.DataFrame:

def delete_rows_by_column_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def sort_dataframe_column_alphabetical_order(df: pd.DataFrame, column_name: str):

def filter_rows_by_column_equals_or_less_than_numeric_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def filter_rows_by_column_strictly_less_than_numeric_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def filter_rows_by_column_equals_or_higher_than_numeric_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def filter_rows_by_column_strictly_higher_than_numeric_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def filter_rows_that_contain_column_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def filter_rows_that_do_not_contain_column_value(
df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:

def exists_value_in_column(df: pd.DataFrame, column: str, value) -> bool:

def count_elements_equal_to_value_in_column(
df: pd.DataFrame, column: str, value: Any = None
) -> int:

def count_elements_containing_value_in_column(
```

```
df: pd.DataFrame, column: str, value: Any = None
) -> int:

def find_n_most_frequent_elements_in_column_subset(
df: pd.DataFrame, target_column: str, subset_column: str, filter_value, n: int
) -> list:

def find_most_frequent_element_in_column_subset(
df: pd.DataFrame, target_column: str, subset_column: str, filter_condition
):

def find_most_frequent_element_in_column(df: pd.DataFrame, column: str = None):

def find_n_most_frequent_elements_in_column(
df: pd.DataFrame, column: str, n: int
) -> list:
```

B. Appendix II: Fuzzy filters

This section contains the code of the functions that implement the fuzzy match filtering in order to explain the steps that this filtering follows.

```
def _best_fuzzy_match(
    series: pd.Series, target: str, threshold: int = 90
) -> Any | None:
    unique_vals = series.dropna().unique()
    best_val, best_score = None, 0
    for v in unique_vals:
        score = fuzz.ratio(str(v).lower(), target.lower())
    if score > best_score:
        best_val, best_score = v, score
        return best_val if best_score >= threshold else None

def filter_rows_that_contain_column_value(
    df: pd.DataFrame, column: str, value: Any = None
) -> pd.DataFrame:
    ...
    threshold = 75
    ...
    # ---- round 1: exact / simple contains --------------------------------
    ...
    # ---- round 2: fuzzy ---------------------------------------------------
    if (
        pd.api.types.is_string_dtype(df[column])
        and isinstance(value, str)
        and value != ""
    ):
        best_match = _best_fuzzy_match(df[column], value, threshold)
        if best_match is not None:
            fuzzy = df[df[column] == best_match]
        if _round_was_useful(original_len, len(fuzzy)):
```

```
            return fuzzy
...
```