DataFrame QA: A Universal LLM Framework on DataFrame Question Answering Without Data Exposure

Anonymous ACL submission

Abstract

This paper introduces DataFrame question answering (QA), a novel task that utilizes large language models (LLMs) to generate Pandas queries for information retrieval and data analysis on dataframes, emphasizing safe and non-revealing data handling. Our method, which solely relies on dataframe column names, not only ensures data privacy but also significantly reduces the context window in the prompt, streamlining information processing and addressing major challenges in LLM-based data analysis. We propose DataFrame QA as a comprehensive framework that includes safe Pandas query generation and code execution. Various LLMs, notably GPT-4, are evaluated using the pass@1 metric on the renowned WikiSQL and our newly developed 'UCI-DataFrameQA', tailored for complex data analysis queries. Our findings indicate that GPT-4 achieves pass@1 rates of 86% on WikiSQL and 97% on UCI-DataFrameQA, underscoring its capability in securely retrieving and aggregating dataframe values and conducting sophisticated data analyses. This approach, deployable in a zero-shot manner without prior training or adjustments, proves to be highly adaptable and secure for diverse applications.

1 Introduction

In the era of large language models (LLMs), table question answering (QA) with LLMs typically involves embedding the entire table into the prompt, along with the user's question and instructions (Li et al., 2023; Chen, 2022). This method is highly effective for querying small and simple tables or dataframes. However, since tables are inherently two-dimensional structures, they can quickly increase the size of the prompt with the addition of rows. This becomes particularly challenging with large dataframes such as those related to weather, traffic, and product sales, which can easily exceed the 4K or 8K content window limit of most models.

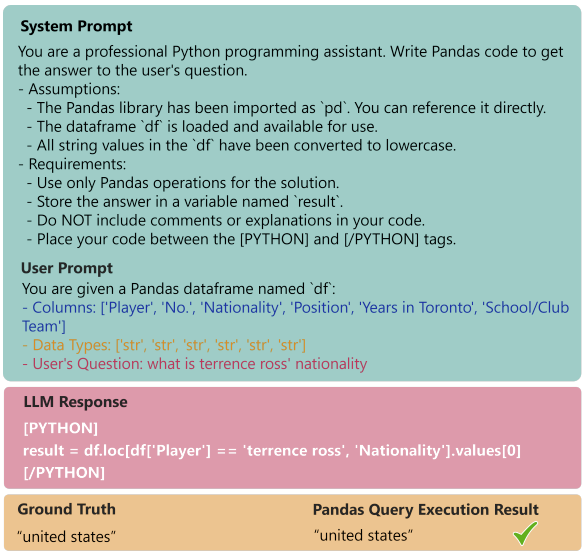

Figure 1: An example where an LLM can generate a correct Pandas query to answer user question using only the table header and column data types, without accessing the table values. Typically, the total number of tokens for DataFrame QA tasks, including both the prompt and the model output, stays below 250 tokens.

Unlike lengthy text content that can be efficiently managed using techniques such as Retrieval-Augmented Generation (RAG) (Lewis et al., 2020), which allows the summarization of each chunk and integrates searching to generate a final answer (thus efficiently processing a large volume of text), large tables present a different challenge. Tables, due to their two-dimensional structure and data density, do not lend themselves to this kind of summarization and retrieval-based processing. This difference highlights the unique challenge of using LLMs to efficiently manage large table data, as opposed to handling extensive text content.

Despite many models (e.g., GPT-4 and Claude-2.1) now expanding their context window sizes to 16k, 32k, or more, several key challenges remain. First, the computational cost for processing tokens is high. For example, it costs $0.01 per 1K tokens

using the latest GPT-4 Turbo model API. Second, embedding full datasets risks potential data leakage. In addition, recent studies indicate the problem of 'Lost in the Middle,' where long prompts can decrease the model performance (Liu et al., 2023). LLMs also struggle with mathematics, which becomes problematic when the query involves calculations (Ouyang et al., 2022). Moreover, conventional querying often includes superfluous data beyond what is necessary to answer the question. These pose challenges in maintaining accuracy, efficiently managing computational resources and protecting sensitive information when applying LLMs to table-based QA.

To address these challenges, we propose a new task and framework called DataFrame QA. This framework aims to enhance the efficiency and security of querying the dataframe by using LLM to generate Pandas queries. It employs a method that utilizes only table column names and data types, effectively reducing data leakage risks and minimizing the need for extensive context windows. We modify the WikiSQL dataset and create a new dataset, UCI-DataFrameQA for our task, conducting evaluations in a zero-shot manner using Llama2 (Touvron et al., 2023), CodeLlama (Roziere et al., 2023), GPT-3.5, and GPT-4 (OpenAI, 2023). Furthermore, our research involves a thorough analysis of the causes of errors and the inherent challenges associated with the DataFrame QA task, along with potential solutions. This investigation provides insights for future dataset expansions and improvements and for enhancing model performance.

2 Task Setups and Challenges

The DataFrame QA task introduces a novel challenge in Natural Language Processing (NLP), focusing on employing LLMs to generate Pandas queries. This highlights the versatility of Pandas, which offers a wide array of data analysis operations, ranging from simple data retrieval to intricate statistical analyses.

2.1 DataFrame QA Task

This task marks a departure from traditional data processing, which typically involves analyzing the relationship between questions and table contents. The DataFrame QA task, instead, concentrates on the analysis of dataframe structures and data types, deliberately omitting the scrutiny of actual data values. It evaluates LLMs' ability to understand dataframe headers and column metadata, and to convert this understanding into valid Pandas queries through natural language questions.

Moreover, the task utilizes the versatility of prompts to include additional information, such as dataset descriptions and specific constraints. This enhances contextual comprehension beyond what traditional fixed-input models offer. This approach facilitates a more dynamic and informed model interaction, greatly expanding the potential of dataframe analysis in NLP.

The input to the DataFrame QA system is defined as a tuple (S, H, C, Q), where S represents supplementary information, like assumptions and Python library constraints, $H = \{h_1, h_2, ..., h_n\}$ denotes the dataframe headers, and $C = \{c_1, c_2, ..., c_n\}$ covers additional column information, such as data types and descriptions. Q is the natural language query. The system's output, P', is a Pandas query generated in response to Q. The ground truth query, P, is the correct Pandas query that yields the answer to Q. Hence, the predicted output is a function of the input:

$$P' = f(S, H, C, Q) \quad (1)$$

$$A' = execute(P', df) \quad (2)$$

$$A = execute(P, df) \quad (3)$$

A' and A are the results obtained by executing Q' and Q on the dataframe df in a safe sandbox, respectively. The effectiveness of the DataFrame QA system is measured by how closely A' approximates the ground truth answer A.

2.2 Challenges

Key challenges of DataFrame QA include:

Interpreting User Questions: Understanding how questions relate to the dataframe's structure and column types, requiring domain-specific knowledge and handling ambiguities.

Formulating Pandas Queries: Proficiency in creating accurate Pandas queries that meet the technical and logical requirements of the task.

Following Instructions: Adhering to given guidelines or constraints, ensuring that responses are technically correct and contextually suitable.

These challenges highlight the intricacies involved in the Dataframe QA task, underscoring the need

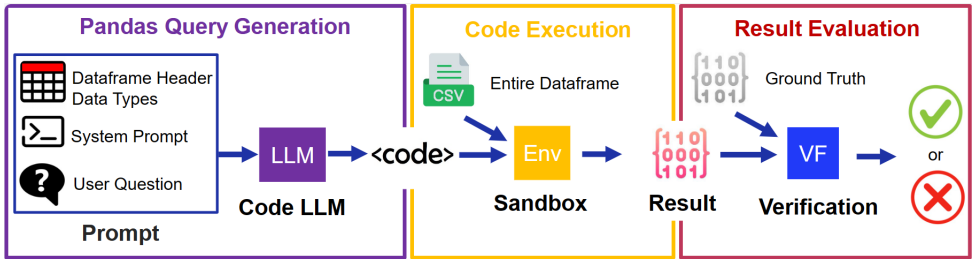

Figure 2: Framework of DataFrame QA.

for LLMs that are not only proficient in natural language processing but also capable of understanding and manipulating complex data structures.

3 Methodology

Figure 2 outlines our DataFrame QA framework, structured in three stages: Pandas Query Generation, Code Execution, and Result Evaluation.

3.1 Pandas Query Generation

In this initial phase, an LLM processes the prompt, comprising the dataframe header, column data types, system prompt (i.e. assumption and requirements), and user question, to generate a Pandas query. This design ensures data privacy by avoiding exposure to table values and leverages column data types to inform query selection.

3.2 Code Execution

The generated query is executed within a controlled virtual environment, protecting against unauthorized operations. This environment is restricted to essential libraries (Pandas, NumPy, and Math), thereby enhancing security. The execution results, stored as Python objects, offer flexibility for further processing. For example, large table results can be provided as downloadable content, while Matplotlib plot objects are displayed directly.

3.3 Result Evaluation

We compare the results of executed queries with ground truth answers, encountering challenges due to the diversity of data types involved. Our methodology standardizes results across numeric, string, and list/ndarray types to facilitate an accurate comparison. Note that Pandas queries often return series or dataframe objects, rather than direct answers to user questions, mirroring the characteristics of coding datasets commonly used in LLM training. To address this, we employ a relaxed evaluation criterion, considering the contents of series or dataframes correct if they include the answer. For pairs where there is a mismatch, we perform a manual comparison to ensure the accuracy and relevance of the results.

4 Experimental Settings

4.1 Dataset

To rigorously assess the proficiency of LLMs in generating Pandas queries for two distinct types of tasks, we have adapted the WikiSQL and UCI datasets to align with our research objectives.

Simple Query Dataset - WikiSQL: WikiSQL (Zhong et al., 2017), a benchmark in Text-to-SQL research, provides a test set comprising 15,878 table-question pairs, designed to evaluate natural language interfaces with relational databases. We transformed these tables into dataframes, ensuring datatype consistency for each column. Additionally, we utilized the results of the SQL queries executed on these tables as the ground truth for our DataFrame QA framework.

A notable challenge in WikiSQL is the frequent lowercasing of entities in user questions, which can lead to ambiguities when formulating Pandas queries. To mitigate this issue, we standardized all user questions and dataframe strings to lowercase. Additionally, we explicitly instructed in the prompt that all strings within the dataframe are lowercased. This approach improves clarity and uniformity in LLM query processing, ensuring consistent interpretation and handling of string data.

The WikiSQL dataset predominantly features straightforward information retrieval questions (71%), solvable with single-step operations similar to basic SQL queries. The remaining 29% focus on aggregation tasks, including 12% MIN/MAX, 9% COUNT, and 8% AVG/SUM. These represent simpler and more direct query scenarios, which require basic dataframe operations.

Complex Dataset - UCI-DataFrameQA: To develop a DataFrame QA dataset reflective of real-

User Question	Pandas Query	Types
which province is bay of islands in?	`result = df.loc[df[‘Electorate’]==‘bay of islands’, ‘Province’].iloc[0]`	Retrieval
how many combined days did go shiozaki have?	`result = df.loc[df[‘Wrestler’]==‘go shiozaki’, ‘Combined days’].values[0]`	Aggregation
how does the average shell weight vary across different numbers of rings?	`result = df.groupby(‘Rings’)[‘Shell_weight’].mean()`	Data Analysis
can you create a new column ‘volume’ as a product of length, diameter, and height, then find the average volume for each sex?	`df[‘Volume’] = df[‘Length’] * df[‘Diameter’] * df[‘Height’];` `result = df.groupby(‘Sex’)[‘Volume’].mean()`	Data Analysis

Table 1: Examples of Sample Questions and Corresponding Pandas Queries Categorized by Complexity Level. Retrieval/Aggregation queries can be resolved using single-step, SQL-like queries, whereas Data Analysis questions necessitate multi-step or complex Pandas operations.

world scenarios, we adopted a comprehensive approach. We sourced diverse dataframes from the UCI dataset (Newman et al., 1998), spanning various domains such as animals, automobiles, and medical fields, to simulate different societal contexts. Our methodology was designed to represent three real-life data interaction roles: **1) Data Scientists**, who delve into detailed data analysis queries for patterns, trends, and statistical insights; **2) General Users**, such as patients in medical datasets or customers in automobile datasets, seeking practical, consumer-oriented aspects of the data; **3) Data Owners**, like hospitals or companies, focusing on extracting business-oriented insights.

Utilizing GPT-4, we generated questions mirroring typical inquiries and challenges these roles face in real-life scenarios, thereby creating a comprehensive DataFrame QA dataset.

For each of the 11 dataframes from the UCI dataset, GPT-4 generated 60 questions (20 per role), each with a corresponding Pandas query. The appendix details the prompts and provides examples of the question/Pandas query generation.

Following a meticulous manual review, we compiled a final set of 547 question/Pandas query pairs. This curation involved eliminating pairs with inaccurate matches, those requiring external libraries, or questions unsolvable with just the provided table headers. This rigorous selection process ensures the dataset comprises realistic and executable DataFrame QA scenarios.

Role	Retrieval/Aggregation	Data Analysis
Data Scientist	9 (5%)	175 (95%)
General User	69 (40%)	105 (60%)
Data Owner	42 (22%)	147 (78%)

Table 2: Distribution of Generated Question Types on UCI Dataset Across Different Roles.

Table 2 presents the distribution of generated question types within the UCI dataset, categorized by different user roles. Analysis of the dataset indicates that 22% of the questions are focused on basic retrieval/aggregation tasks, while a predominant 78% involves more advanced operations, such as grouping, correlation analysis, and sorting. Specifically, the Data Scientist role primarily concerns complex data analysis queries, whereas General Users tend to concentrate on more straightforward retrieval/aggregation questions. Questions from Data Owners exhibit a range of complexity, bridging the two extremes. The dataset is a pivotal testbed for assessing LLMs' adeptness in handling advanced queries, showing their capacity to execute complex, multi-step data analysis.

This distinction in question complexity across two datasets provides an opportunity to evaluate LLMs' capabilities over a wide spectrum of query complexities, ranging from simple data retrieval to sophisticated data manipulation tasks.

4.2 Baselines

Our experimental baselines include:

Llama2. This advanced iteration of the Llama language model series offers configurations ranging from 7B to 70B parameters (Touvron et al., 2023). With training on 2 trillion tokens, it is equipped with an expanded 4K token context window, enhancing its applicability across diverse NLP tasks.

CodeLlama. A specialized variant of Llama2, CodeLlama is tailored for coding-related tasks (Roziere et al., 2023). It demonstrates superior performance in coding benchmarks, benefitting from a 16K token window. We utilized its instruction models at 7B, 13B, and 34B parameter sizes, focusing on their ability to generate and interpret code.

GPT-3.5, GPT-4. These models are benchmarks in the LLM domain, showcasing exceptional performance across coding, general NLP tasks, and conversational capabilities. Their ongoing updates

reinforce their status as leaders in AI-driven solutions. The model versions used in our experiments are gpt-3.5-turbo-0613 and gpt-4-0613.

4.3 Implementation Details

In our methodology, a consistent greedy decoding strategy was applied across all LLMs. For the deployment of Llama2 and CodeLlama, the official checkpoints available through the HuggingFace were used. The models were executed on NVIDIA A100 GPUs, with the number of units ranging from one to four, depending on the model size.

4.4 Evaluation Metric

The pass@1 score (Chen et al., 2021) is a crucial metric in evaluating the performance of LLM in the context of code generation tasks. It measures the accuracy of the LLMs in providing the correct answer on their first attempt without any additional iterations or refinements. This score is especially important in scenarios where immediate and precise responses are required, reflecting the model's ability to accurately interpret and respond to complex queries in a single pass.

5 Experimental Analysis

This study delves into the DataFrame QA task, focusing on the capabilities and challenges of cutting-edge LLMs. Our investigation is centered around three pivotal research questions: **Q1. Are current state-of-the-art LLMs capable of effectively handling the DataFrame QA task? Q2. What factors influence the performance differences among various LLMs? Q3. What are the inherent challenges and potential solutions associated with DataFrame QA tasks?**

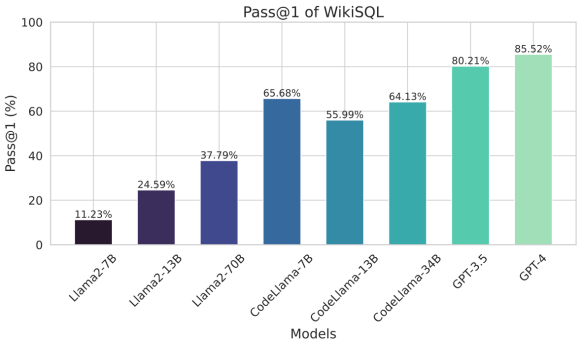

Figure 3: Performance of LLMs on WikiSQL.

Q1: Efficacy of Leading LLMs in DataFrame QA Tasks Our study assesses various LLMs' first-attempt accuracy (pass@1), with findings illustrated in Figures 3 and 4. Notably, GPT-4 exhibits high pass@1 accuracies—85.5% on WikiSQL and 97.2% on UCI-DataFrameQA, reflecting its adeptness in processing a wide array of queries, from simple data retrieval to complex analysis, without direct access to table data.

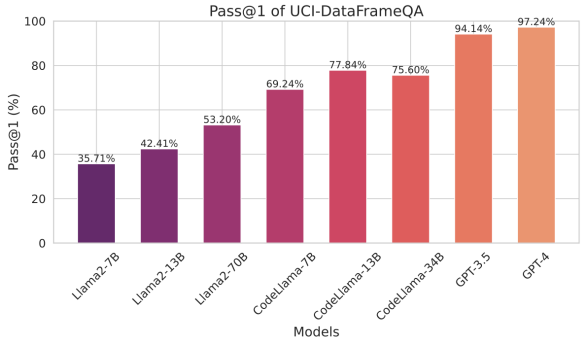

Figure 4: Performance of LLMs on UCI-DataFrameQA.

Analysis of GPT-4 on WikiSQL's test set reveals that 11.6% of queries pose challenges due to mismatched pairs (5%), ambiguity (5%), and quotation mark issues (1.6%), affecting execution. Addressing these issues, GPT-4's pass@1 accuracy could rise to 96.7%, aligning with its performance on UCI-DataFrameQA.

Scaling Laws in LLM Performance: We observed clear performance stratification among models: Llama2 and CodeLlama, with the latter surpassed by the GPT series. GPT and Llama2 models adhere to scaling laws ((Kaplan et al., 2020)), showing gains with increasing size. In contrast, CodeLlama models deviate from these laws, warranting further exploration in subsequent sections.

Comparison with Text-to-SQL Models: GPT-4, in comparison to specialized Text-to-SQL models such as TAPEX (Liu et al., 2021) with an execution accuracy of 89.5% and SeaD+EG$_{CS}$ (Xu et al., 2021) at 92.7%, exhibits slightly lower performance. This discrepancy is attributed to two main factors: **Task Complexity:** Generating Pandas queries is inherently more complex than structured SQL queries, given Pandas' wider operation range. **Zero-Shot Learning Approach:** Unlike TAPEX and SeaD+EG$_{CS}$, which are specialized for Text-to-SQL, GPT-4's zero-shot application, without specific fine-tuning, impacts its efficiency in DataFrame QA tasks.

Q2. Performance Variation Determinants

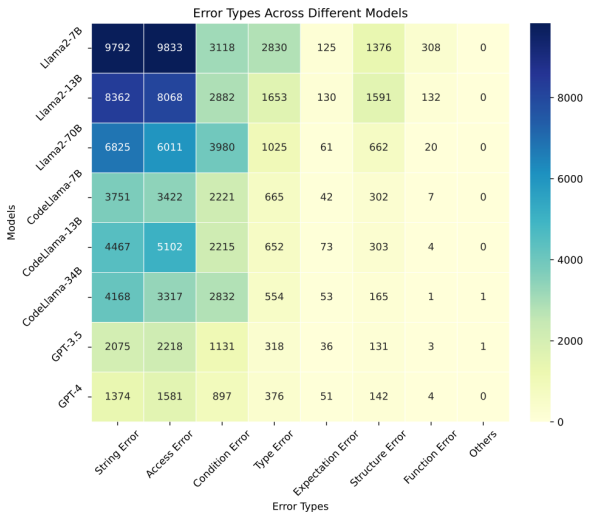

Figure 5: Distribution of error types among different LLMs on WikiSQL. Definitions of error types and prompts for error classification are put in Appendix.

In delving into what drives performance differences in LLMs, we scrutinize failure cases on WikiSQL using GPT-3.5. Starting with 100 error samples, we categorized these using GPT-4 and further confirmed through manual verification, identifying eight distinct error types detailed in the Appendix. We then used GPT-3.5 to classify all error classes across model input, sample rows, generated queries, execution output, and expected results, where incorrect queries often span multiple error categories.

Error Distribution Across LLMs: Figure 5 presents a heatmap of error class distribution across LLMs. Predominant errors include String Matching and Comparison, Data Access and Bounds, and Query Condition and Value. These stem from issues like misidentified column names, case sensitivity, and misinterpretations of user questions and instructions, often due to a mismatch between instructions and their execution.

Bias Between Instructions and Training Data: In our analysis of CodeLlama models, we identified a notable bias reflecting discrepancies between instruction adherence and training data distribution, particularly evident in handling string queries. The 13B model registered 2368 instances, and the 34B model recorded 1175 instances of errors related to this bias, in stark contrast to the mere 175 instances encountered by the 7B model. These errors persisted even with explicit instructions to treat all strings as lowercase, underscoring a variance in the models' interpretation and compliance with these directives, deviating from the expected scaling law behavior in such tasks.

For instance, `df[df['Player']=='Terrence Ross']['Nationality'].values[0]` incorrectly queries the capitalized `'Terrence Ross'`, contrary to the directive of all strings in the dataframe being lowercase. This example highlights the bias issue: when querying dataframe values, especially with proper nouns like names and places, the model tends to capitalize them, showing a preference influenced by its training data over the provided instructions. This tendency exemplifies the broader challenge of aligning model behavior with specific directives, particularly when they counter the model's training data distribution.

These results reveal that DataFrame QA tasks test not only the coding abilities of LLMs but also their ability to adhere to instructions in the face of conflicts with training data distribution, along with their comprehensive skills in understanding user queries and dataframe headers.

Q3. Inherent Challenges and Solutions in DataFrame QA Tasks

In Table 3, we outline typical failure cases in DataFrame QA tasks, categorizing a spectrum of error types commonly encountered by LLMs. These include Value Retrieval, Column Reference, Instruction Misalignment, Aggregation, Function-Column Ambiguity, Insufficient Column Data/Format Information, Coding Syntax, and Hallucination Errors, each posing distinct challenges. Detailed examples and solutions for varied failure types are comprehensively provided in the Appendix.

Value Retrieval Error: This error occurs when Pandas queries fetch incorrect values due to ambiguous user questions. For instance, a query for `'grey and bell'` might be misinterpreted as separate searches for `'grey'` and `'bell'`. Also, LLMs might omit crucial characters like Roman numeral `'I'` or hyphens, mistaking them for textual errors. *Solution:* Utilizing quotation marks around specific query terms can significantly reduce these errors, clarifying the intended search as a singular entity.

Column Reference Error: Errors occur when queries incorrectly target columns, often due to ambiguous names. This common challenge arises especially when LLMs lack domain-specific knowledge, leading to column misidentification. *Solution:* Providing clear column descriptions in prompts, such as specifying the roles of `'ca'` and `'num'` columns, can greatly reduce these errors.

Table 3: Typical Failure Cases in DataFrame QA Task.

Error Type	User Question	Generated Pandas Query	Expected Pandas Query
Value Retrieval Error	which province is grey and bell electorate in	result = df[(df[‘Electorate’]==**‘grey’**)| (df[‘Electorate’]==**‘bell’**)][‘Province’].unique()	result=df[(df[‘Electorate’]==**‘grey and bell’**)] [‘Province’].unique()
Column Reference Error	what is the variance in resting blood pressure (trestbps) among different heart disease severity levels?	df.groupby(**‘ca’**)[‘trestbps’].var()	df.groupby(**‘num’**)[‘trestbps’].var()
Instruction Misalignment	What are the mean and median lengths of abalone by each sex category?	**import pandas as pd** **# Group the dataframe by sex** ...	result = df.groupby(‘Sex’)[‘Length’].agg([‘mean’, ‘median’])
Aggregation Error	what is the total amount of allied-unrelated where the component is human capital?	result = df[df[‘Component’]==‘human capital’] [‘Allied-Unrelated’].**sum()**	result = df[df[‘Component’]==‘human capital’] [‘Allied-Unrelated’].**count()**
Function-Column Ambiguity Error	what is the average score when the swimsuit score is 8.503	result = df[df[‘Swimsuit’]==‘8.503’] [‘Average’].**mean()**	result = df[df[‘Swimsuit’]==‘8.503’][‘Average’]
Insufficient Column Data/Format Information	which team played on december 5?	df[‘Date’] = pd.**to_datetime**(df[‘Date’]) result = df[df[‘Date’]==‘December 5’][‘Team’]	result = df[df[‘Date’] == ‘december 5’][‘Team’]
Coding Syntax Error	how does the average shell weight vary across different numbers of rings?	result = df.groupby(‘Rings’).**mean()** [‘Shell_weight’]	result = df.groupby(‘Rings’)[‘Shell_weight’] .**mean()**
Hallucination Error	i’m interested in knowing the most common age of abalone. can you find that for me?	result = df[**‘Age’**].mode()	result = df[**‘Rings’**].mode()]

Instruction Misalignment Error: This error occurs when LLMs deviate from given instructions, often seen in superfluous import statements and comments, typical in training datasets but unnecessary for DataFrame QA tasks. It also includes issues like case sensitivity errors, where LLMs incorrectly handle capitalized names or terms.

Solution: Precise directives in prompts, such as `‘Do not import Pandas library’` and specific case sensitivity guidelines, help ensure LLMs’ adherence to task-specific requirements.

Aggregation Error: These errors occur when LLMs apply incorrect aggregation functions, often because questions contain words common to both operations and column names, like `‘average’`.

Solution: Clear column information and specific query formulations, such as stating the nature of `‘allied-unrelated’` columns, guide LLMs to apply the correct aggregation method.

Function-Column Ambiguity Error: This type of error manifests when there is confusion between column names and function names, leading to incorrect query execution.

Solution: Renaming columns may not always work. Encapsulating column names in quotes in queries can distinguish them from function commands, aiding accurate interpretation.

Insufficient Column Data/Format Information Error: These errors often occur due to mismatches between the LLM’s assumptions about dataset structure and the actual data format, particularly in handling date-related queries.

Solution: Specifying column formats, such as the format of `‘Date’` column, in prompts ensures precise LLM data handling.

Coding Syntax Error: Highlights differences in LLMs’ coding capabilities, especially in structuring and executing dataframe queries.

Solution: Choosing an advanced base LLM or training on DataFrame QA datasets enhances their query optimization and data handling skills.

Hallucination Error: This type of error arises when LLMs create responses based on incorrect assumptions or non-existent data, often due to a lack of domain knowledge.

Solution: Providing detailed data and column information in prompts, like explaining how abalone age is determined, helps LLMs bridge domain knowledge gaps and improve query accuracy.

In summary, DataFrame QA tasks present inherent challenges that broadly fall into two categories. The first pertains to the inherent capabilities of LLMs, particularly visible in issues like Instruction Misalignment and Coding Syntax Errors. GPT models, in particular, have a significantly lower error rate in these areas compared to other models, showcasing their superior ability to align with human instructions and coding accuracy.

The second category comprises challenges unique to the DataFrame QA environment. This includes complexities in question interpretation, table header naming, and domain knowledge gaps, which are central to the task-specific intricacies. Additionally, difficulties that arise from missing table value formats and value range specifications also contribute to this category, leading to errors

in query processing. Tackling both the model-intrinsic limitations and these dataframe-specific complexities is vital for enhancing LLMs' performance in DataFrame QA tasks.

6 Related Work

Table QA is a domain of natural language processing that focuses on interpreting and answering queries based on tabular data. This field can be broadly divided into two key tasks:

6.1 Text-to-SQL

This task involves converting natural language questions into SQL queries that can be executed against relational databases. The aim is to accurately interpret the user's intent and translate it into syntactically and semantically correct SQL commands. Recent advancements in Text-to-SQL have primarily leveraged neural network-based approaches, including LLMs (Ye et al., 2023; Ni et al., 2023), especially sequence-to-sequence models (Liu et al., 2021; Xu et al., 2021; Herzig et al., 2020; Yu et al., 2018; Zhong et al., 2017). These technologies have demonstrated significant effectiveness in understanding diverse queries across various domains and in generating the corresponding SQL statements. Current Text-to-SQL technologies primarily rely on simple database schemas and basic queries, limiting their ability to handle complex, real-world database structures and advanced relational tasks.

6.2 QA on Semi-Structured Tables

This task focuses on accurately parsing HTML tables, which are often semi-structured and vary in format, to understand the context and relationships within the data and provide the correct answers (Pasupat and Liang, 2015). It requires advanced techniques in data extraction, contextual understanding, and natural language processing to effectively navigate the diverse structures and formats of HTML tables. Recent developments have utilized transformer-based models (Xie et al., 2022; Pan et al., 2021; Glass et al., 2021; Yin et al., 2020) and LLMs (Li et al., 2023), significantly improving the ability to process and interpret complex table structures and query contexts. These models have notably improved the accuracy and efficiency of extracting information from semi-structured HTML tables, representing a substantial advancement in the field. However, these models face limitations when loading HTML/CSV-formatted tables, often showing limited proficiency in tasks such as identifying missing cells or finding column names, leading to low accuracy in specific tests (Li et al., 2023). Including table values in input poses data privacy risks and challenges due to context window limitations and the handling of sensitive information.

7 Conclusions

In summary, our introduction of a new DataFrame QA task and framework represents a significant advancement in the field. This zero-shot approach, which leverages dataframe headers and datatypes along with user questions and deliberately excludes table values, addresses data privacy concerns and minimizes extraneous data in prompts. Beyond this, DataFrame QA can further enrich the prompts with dataset descriptions and column data format details, aiding in clarifying the column meanings within dataframes. This method not only offers improved control over code execution outputs but also provides greater scalability compared to traditional Text-to-SQL tasks.

Through testing with advanced open-source and closed-source LLMs, we have analyzed error patterns, challenges and determined that the efficacy of DataFrame QA relies not only on the coding abilities of LLMs but also on their understanding of the relationship between user questions, dataframe columns, and provided instructions. In particular, the accuracy rate of GPT-4 largely consistent with practical applications.

8 Limitation and Future Work

A current limitation of our DataFrame QA framework is its restriction to the Pandas library, lacking integration with other relevant libraries such as NumPy, scikit-learn, and Matplotlib. Moving forward, our future work aims to address this gap by expanding the variety of libraries utilized, investigating the synergistic capabilities of these libraries to solve more complex problems. Additionally, we plan to enhance our framework to support a multi-agent system, which could add significant depth and functionality. Another key area of future development involves expanding our dataset, transforming the DataFrame QA into a task that better trains LLMs in coding capabilities. This expansion will not only aim to bridge gaps in interpreting real-world questions and generating code but will also explore the inclusion of questions involving numerical calculations and statistics.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen. 2022. Large language models are few (1)-shot table reasoners. *arXiv preprint arXiv:2210.06710*.

Michael Glass, Mustafa Canim, Alfio Gliozzo, Saneem Chemmengath, Vishwajeet Kumar, Rishav Chakravarti, Avi Sil, Feifei Pan, Samarth Bharadwaj, and Nicolas Rodolfo Fauceglia. 2021. Capturing row and column semantics in transformer based question answering over tables. *arXiv preprint arXiv:2104.08303*.

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2023. Table-gpt: Table-tuned gpt for diverse table tasks. *arXiv preprint arXiv:2310.09263*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2021. Tapex: Table pre-training via learning a neural sql executor. *arXiv preprint arXiv:2107.07653*.

D.J. Newman, S. Hettich, C.L. Blake, and C.J. Merz. 1998. Uci repository of machine learning databases.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

OpenAI. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Feifei Pan, Mustafa Canim, Michael Glass, Alfio Gliozzo, and Peter Fox. 2021. Cltr: An end-to-end, transformer-based system for cell level table retrieval and table question answering. *arXiv preprint arXiv:2106.04441*.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I Wang, et al. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *arXiv preprint arXiv:2201.05966*.

Kuan Xu, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. 2021. Sead: End-to-end text-to-sql generation with schema-aware denoising. *arXiv preprint arXiv:2105.07911*.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 174–184.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. *arXiv preprint arXiv:1810.05237*.

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

Appendices

A UCI-DataFrameQA Dataset Generation with GPT-4

B Error Classification with GPT-3

C Examples of Challenges in DataFrame QA and Potential Solutions

You are given a dataframe and are tasked with generating real-world questions and corresponding Pandas queries for specific roles (customer, data analyst, and company owner, etc.). The dataframe is described as follows:

- Name of dataframe: Automobile
- Description of dataframe: This data set consists of three types of entities: (a) the specification of an auto in terms of various characteristics, (b) its assigned insurance risk rating, (c) its normalized losses in use as compared to other cars. The second rating
... *(omit dataset description to save space)*

- Column information of dataframe:
```
#   Column            Dtype
--- ------            -----
0   normalized-losses  float
1   make               str
2   fuel-type          str
```
... *(omit column information to save space)*

- Sample rows of dataframe in CSV format:
```
normalized-losses,make,fuel-type,aspiration,num-of-doors,body-style,drive-wheels,engine-location,wheel-base,length,width,height,curb-weight,engine-type,num-of-cylinders,engine-size,fuel-system,bore,stroke,compression-ratio,horsepower,peak-rpm,city-mpg,highway-mpg,price,symboling
134.0,toyota,gas,std,2.0,hardtop,rwd,front,98.4,176.2,65.6,52.0,2540,ohc,4,146,mpfi,3.62,3.5,9.3,116.0,4800.0,24,30,8449.0,2
168.0,toyota,gas,std,2.0,hatchback,rwd,front,94.5,168.7,64.0,52.6,2204,ohc,4,98,2bbl,3.19,3.03,9.0,70.0,4800.0,29,34,8238.0,1
```
... *(omit sample rows of dataframe to save space)*

- Task:
Using the provided information about the dataframe, formulate 10 natural language questions for the data scientist category.

- Characteristics of the questions from data scientist category:
Questions tailored for individuals with an in-depth understanding of the dataset, possessing expertise in statistical and mathematical analysis. These questions should challenge their analytical skills, encouraging the use of advanced data manipulation and interpretation techniques. The focus is on extracting complex insights and patterns from the data.

- Guidelines:
- All questions must be solvable using the Pandas library in Python.
- Questions should encompass a wide range of Pandas operations, from basic to advanced functionalities.
- Questions must reflect the real-world interests of the specified role.

- Assumptions:
- The Pandas library has been imported as `pd`. You can reference it directly.
- The dataframe `df` is loaded and available for use.

- Response:
- Store each answer in a variable named `result`.
- Do NOT include comments or explanations in your response.
- Present your questions and code within the list.

- Expected Response Format:
```python
[
   {
      "question": "...",
      "query": "result = ",
   },
   {...},
   ...
]
```

Figure 6: **Sample Prompt for DataFrame QA Dataset Generation with GPT-4.**

Table 4: Characteristics of Questions from Three Different Roles Used in the Prompt.

Role	**Description of Question Characteristics**
Data Scientist	Questions tailored for individuals with an in-depth understanding of the dataset, possessing expertise in statistical and mathematical analysis. These questions should challenge their analytical skills, encouraging the use of advanced data manipulation and interpretation techniques. The focus is on extracting complex insights and patterns from the data.
General User	Questions designed for users who may not have specialized data analysis skills but are interested in the practical, consumer-oriented aspects of the data. These questions should be formulated based on the nature and context of the data, requiring inferential thinking about its potential end-users. Questions and queries should be structured to be somewhat open-ended, avoiding direct references to specific column names, thus introducing a level of interpretative ambiguity.
Data Owner	Questions aimed at individuals or entities who own or have created the data, with a focus on business-oriented insights. These questions should cater to their interest in understanding the broader business implications, trends, and strategic insights that can be derived from the data. The emphasis is on leveraging the data for decision-making, performance tracking, and identifying opportunities or areas for improvement within the business context.

Table 5: Sample Generated Question/Pandas Query Pairs of UCI-DataFrameQA Dataset.

Role	**Question**	**Pandas Query**
Data Scientist	How has the average weight of cars changed over the model years?	`result = df.groupby('model_year')['weight'].mean()`
Data Scientist	What is the distribution of tumor size for cases with recurrence events?	`result = df[df['Class'] == 'recurrence-events'] ['tumor-size'].value_counts()`
General User	Which cars have more than 6 cylinders?	`result = df[df['cylinders'] > 6]`
General User	What is the most common tumor size observed in the data?	`result = df['tumor-size'].mode()[0]`
Data Owner	What are the names of the cars with the top 3 highest fuel efficiencies in our dataset?	`result = df.nlargest(3, 'mpg')['car_name']`
Data Owner	What is the frequency of tumor sizes in the age group 50-59?	`result = df[df['age'] == '50-59']['tumor-size'] .value_counts()`

Table 6: Description of Eight Pre-defined Error Classes.

Abbreviation	**Error Classes**	**Description**
String Error	String Matching and Comparison Errors	Errors in this class arise from improper handling of string comparisons, such as failing to use appropriate matching methods, not accounting for case sensitivity, whitespace, or special characters, and using exact matching where pattern recognition is required.
Access Error	Data Access and Bounds Errors	This class is for errors when data is accessed using an incorrect index or key, or when the index exceeds the bounds of the data structure, leading to `'index out of bounds'` or `'key not found'` errors.
Condition Error	Query Condition and Value Errors	This class covers errors where query conditions do not reflect the data accurately or the wrong values are used, resulting in no matches or incorrect results. It includes using incorrect column names or values and failing to match the query criteria with the dataset.
Type Error	Data Type and Operation Errors	This class includes errors from attempting operations between incompatible data types, using methods unsuitable for the data type, and applying aggregation functions incorrectly, often leading to type mismatches or operation errors on non-compatible data types.
Expectation Error	Expectation and Interpretation Errors	This class encompasses errors from a discrepancy between expected outcomes and actual results, which may stem from misinterpreting the output, data, or having incorrect expectations of the data's structure, leading to incorrect assumptions and results.
Structure Error	Data Structure Reference Errors	This class refers to errors arising from incorrect assumptions or references to the data's structure, such as referencing non-existent columns or misinterpreting the content of the data, leading to queries that do not align with the actual data format or content.
Function Error	Function and Method Usage Errors	Errors in this category result from misusing functions or methods outside their intended purpose, such as using a function designed for a specific operation in a context where it does not apply, or calling methods on objects they are not designed for.
Others	Others	The category to cover any errors that do not fit into the specific categories above, such as general mistakes in code logic or implementation that leads to unexpected results or errors.

- Task Description:
I am trying to process a DataFrame with Python to find out the answer to the question. My code is not producing the expected result, and I need assistance to identify the error type.

- Question:
how many schools or teams had jalen rose

- Sample Rows from DataFrame `df`:
,Player,No.,Nationality,Position,Years in Toronto,School/Club Team
0,aleksandar radojević,25,serbia,center,1999-2000,barton cc (ks)
1,shawn respert,31,united states,guard,1997-98,michigan state
2,quentin richardson,n/a,united states,forward,2013-present,depaul
3,alvin robertson,"7, 21",united states,guard,1995-96,arkansas
4,carlos rogers,"33, 34",united states,forward-center,1995-98,tennessee state

- My Code:
result = df.loc[df['Player'] == 'Jalen Rose', 'School/Club Team'].nunique()

- Actual Output of My Code:
0

- Expected Output:
1.0

- Error Class and Description
Error Class	Description
Query Condition and Value Errors	This class covers errors where query ...
...
(This table is omitted to save space. Details can be found in Table 6)

- Assumption:
- All the string values in the `df` are converted to lowercase.
- All the words in the question are converted to lowercase.
- I can only access the dataframe column names when I write the Python code. I don't have access to the data itself. So my code may not be correct if the question is unclear or the column names are unclear.

- Requirement:
- Classify the error type in my code from the error class table above. Choose all error classes from above and place your answer between the [ERROR] and [/ERROR] tags. For example:
[ERROR]
Query Condition and Value Errors
[/ERROR]

- Error could belong to multiple error classes.
- Provide a concise justification for your error classification decision. Place your answer between the [EXPLANATION] and [/EXPLANATION] tags. For example:
[EXPLANATION]
Your explanation
[/EXPLANATION]

- Don't include comments or explanations in your answer.

Figure 7: **Sample Prompt for Error Classification with GPT-3.** Provided are the question, sample rows in the target dataframe, generated Pandas query, execution result, ground truth answer, and 8 pre-defined error classes with definitions. GPT-3 is tasked with classifying the errors present in this example. In this case, the incorrect Pandas query generated by CodeLlama-34B erroneously capitalizes the player's name when querying the dataframe.

[ERROR]
Query Condition and Value Errors
String Matching and Comparison Errors
[/ERROR]

[EXPLANATION]
The error in the code is due to a combination of query condition and value errors and string matching and comparison errors.

In the code, the query condition `df['Player'] == 'Jalen Rose'` is used to filter the DataFrame based on the 'Player' column. However, the actual value in the 'Player' column is in lowercase, while the query condition is in title case. This results in no matches being found and the output being 0.

Additionally, the string matching and comparison error arises from not accounting for case sensitivity. The code is looking for an exact match of 'Jalen Rose', but the actual value in the DataFrame may be 'jalen rose' or 'JALEN ROSE' due to the assumption that all string values are converted to lowercase.

To fix the error, the query condition should be modified to `df['Player'].str.lower() == 'jalen rose'` to ensure case-insensitive matching.
[/EXPLANATION]

Figure 8: **Sample Response for Error Classification with GPT-3.** We demonstrate the outcome of employing GPT-3.5 to assist in identifying the type of error. The response accurately categorizes the error, identifying it as a String Matching and Comparison Error, as well as a Query Condition and Value Error.
The analysis by GPT-3.5 underscores the importance of using lowercase search criteria for proper nouns (e.g., `'jalen rose'` instead of `'Jalen Rose'`), and recommends the implementation of the `.str.lower()` function as a standard practice. This approach ensures uniform conversion of strings to lowercase within dataframes, thereby mitigating potential case sensitivity issues. This specific case serves as a prominent example of the effectiveness of LLMs in conducting detailed error classification and analysis, demonstrating their significant value in addressing complex data processing challenges in natural language processing research.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Member', 'Electorate', 'Province', 'MPs term', 'Election date']
- Data Types: ['str', 'str', 'str', 'str', 'str']
- User's Question: which province is grey and bell electorate in

<Generated Pandas Query>
result = df[(df['Electorate'] == 'grey') | (df['Electorate'] == 'bell')]['Province'].unique() ❌

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Member', 'Electorate', 'Province', 'MPs term', 'Election date']
- Data Types: ['str', 'str', 'str', 'str', 'str']
- User's Question: which province is **'grey and bell'** electorate in

<Generated Pandas Query>
result = df.loc[df['Electorate'] == 'grey and bell', 'Province'].iloc[0] ✔

Figure 9: **Value Retrieval Error by GPT-4.** This type of error occurs in pandas queries when an incorrect value is retrieved. It often arises due to ambiguities in the user's question, leading to multiple possible interpretations. For instance, consider the example where the query targets the term `'grey and bell'` While the intended search might be for a combined entity, `'grey and bell'`, it could also be misinterpreted as two separate searches for `'grey'` and `'bell'` respectively. Another common occurrence in this error category involves GPT models inadvertently omitting special characters or symbols, such as Roman number `'I'` or hyphens, during query execution. This usually happens because the model mistakenly identifies these characters as typographical errors in the sentence, rather than integral parts of the search value. This represents one of the most common error categories in DataFrame QA tasks.
Solution: Enclosing query terms in quotation marks can significantly reduce Value Retrieval Errors. For instance, using quotations to specify `'grey and bell'` as a single entity, aiding in precise and accurate value retrieval.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'num']
- Data Types: ['int', 'int', 'int', 'int', 'int', 'int', 'int', 'int', 'int', 'float', 'int', 'float', 'float', 'int']
- User's Question: what is the variance in resting blood pressure (trestbps) among different heart disease severity levels?

<Generated Pandas Query>
result = df.groupby('ca')['trestbps'].var() ❌

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'num']
- Data Types: ['int', 'int', 'int', 'int', 'int', 'int', 'int', 'int', 'int', 'float', 'int', 'float', 'float', 'int']
- Column 'ca' includes the number of major vessels (0-3) colored by flourosopy.
- Column 'num' includes the diagonals of heart disease.
- User's Question: what is the variance in resting blood pressure (trestbps) among different heart disease severity levels?

<Generated Pandas Query>
result = df.groupby('num')['trestbps'].var() ✔

Figure 10: **Column Reference Error by GPT-4**. This error manifests when a query erroneously targets an incorrect column, commonly attributable to inadequately defined or ambiguous column names. Predominant in DataFrame QA tasks, this error underscores the imperative for a comprehensive understanding of table headers to facilitate precise data retrieval. LLMs, lacking prior domain-specific knowledge, are particularly susceptible to misidentifying columns, thereby yielding inaccurate outcomes. In the above example, the LLM fails to discern the significance of the `'num'` column, which is indicative of the diagnosis of heart disease. This oversight underscores the pivotal role of appropriate table header naming in ensuring the accuracy of DataFrame QA tasks.
Solution: Clarifying ambiguous columns in prompts can greatly reduce Column Reference Errors. For instance, stating "`Column 'ca' represents major vessels colored by fluoroscopy`" and "`Column 'num' indicates heart disease diagnoses`" guides LLMs to the correct data, enhancing query accuracy.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- User's Question: what are the mean and median lengths of abalone by each sex category?

<Generated Pandas Query> ❌

```
import pandas as pd

# Group the dataframe by sex
grouped_df = df.groupby('Sex')

# Calculate the mean and median lengths for each sex category
mean_lengths = grouped_df['Length'].mean()
median_lengths = grouped_df['Length'].median()

# Store the results in a dictionary
result = {'Mean Lengths': mean_lengths, 'Median Lengths': median_lengths}
```

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - **Do not import Pandas library.**
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- User's Question: what are the mean and median lengths of abalone by each sex category?

<Generated Pandas Query> ✔

```
result = df.groupby('Sex')['Length'].agg(['mean', 'median'])
```

Figure 11: **Instruction Misalignment Error by CodeLlama-7B.** This error type emerges when an LLM fails to follow or comprehend given instructions. A case in the above figure, despite explicit instructions in the prompt that the pandas library can be directly utilized and that the output should not include comments, the LLM deviates from this guideline. Case sensitive cases discussed in this paper also belong to this class. This indicates a misalignment with the provided instructions, reflecting a potential bias or conflict stemming from the LLM's training on datasets where import statements and comments are standard. In our specific DataFrame QA task, such inclusions are redundant and contrary to the task requirements. This scenario exemplifies the importance of an LLM's ability to adapt to the specific nuances and requirements of a given task, distinguishing between standard programming practices and task-specific directives.

Solution: Enhancing prompts with clear directives can effectively prevent Instruction Misalignment Errors. For example, specifying `'Do not import Pandas library'` alongside `'Pandas is pre-imported as pd'` emphasizes the need for LLMs to strictly follow given instructions, improving task adherence and accuracy.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Component', 'Integrated', 'Allied-Related', 'Allied-Unrelated', 'Holding']
- Data Types: ['str', 'str', 'str', 'str', 'str']
- User's Question: what is the total amount of allied-unrelated where the component is human capital?

<Generated Pandas Query>
result = df[df['Component'] == 'human capital']['Allied-Unrelated'].sum() ❌

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Component', 'Integrated', 'Allied-Related', 'Allied-Unrelated', 'Holding']
- Data Types: ['str', 'str', 'str', 'str', 'str']
- Column 'Allied-Unrelated' has categorical values.
- User's Question: what is the total amount of allied-unrelated where the component is human capital?

<Generated Pandas Query>
result = df[df['Component'] == 'human capital']['Allied-Unrelated'].count() ✔

Figure 12: **Aggregation Error by GPT-4.** This error arises in dataframe queries when an LLM incorrectly applies an aggregation function due to a lack of clear understanding of the specific meaning and content of a column, compounded by the ambiguity of the question. The crux of the error lies in the LLM's inability to distinguish whether a query requires the use of a summation (sum) or a count function. For instance, in response to a query like `'What is the total amount under certain conditions?'`, it's vital to discern whether the query seeks the sum of all data meeting the criteria (using the sum() function) or merely the number of instances that satisfy the conditions (using the count() function). If an LLM does not fully grasp the nuances of the column's specific meaning and the data characteristics, or if it fails to accurately interpret the intent of the question, it may select an inappropriate aggregation function, leading to results that do not align with the actual requirements.
Solution: Providing clear column information and specific query formulations can effectively prevent Aggregation Errors. For instance, stating "`Column 'allied-unrelated' holds categorical data`" or rephrasing a query to "`Count 'allied-unrelated' entries for human capital`" guides the LLM to apply the correct aggregation method, enhancing result accuracy.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['State', 'Preliminaries', 'Interview', 'Swimsuit', 'Evening Gown', 'Average']
- Data Types: ['str', 'str', 'str', 'str', 'str', 'str']
- User's Question: what is the average score when the swimsuit score is 8.503?

<Generated Pandas Query>
result = df[df['Swimsuit'] == '8.503']['Average'].mean() ❌

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['State', 'Preliminaries', 'Interview', 'Swimsuit', 'Evening Gown', 'Average']
- Data Types: ['str', 'str', 'str', 'str', 'str', 'str']
- User's Question: what is the **'average'** score when the swimsuit score is 8.503?

<Generated Pandas Query>
result = df[df['Swimsuit'] == '8.503']['Average'] ✔

Figure 13: **Function-Column Ambiguity Error by GPT-4.** This error manifests in dataframe queries when there is ambiguity between column names and function names, leading to erroneous interpretations and executions by the LLM. A typical instance of this error is seen when a column name includes terms like `'Average'` or `'Avg'` which can also be names of aggregation functions. This leads to an unnecessary aggregation operation instead of the simple data retrieval that the query required.
Solution: Renaming columns, such as changing to `'average score'`, may not rectify Function-Column Ambiguity Errors. A preferable approach is to encapsulate column names in quotation marks within queries, clearly differentiating them from function commands, thereby guiding accurate LLM interpretation.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Game', 'Date', 'Team', 'Score', 'High points', 'High rebounds', 'High assists', 'Location Attendance', 'Record']
- Data Types: ['float', 'str', 'str', 'str', 'str', 'str', 'str', 'str', 'str']
- User's Question: which team played on december 5?

<Generated Pandas Query>
df['Date'] = pd.to_datetime(df['Date'])
result = df[df['Date'] == 'december, 5']['Team']

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Game', 'Date', 'Team', 'Score', 'High points', 'High rebounds', 'High assists', 'Location Attendance', 'Record']
- Data Types: ['float', 'str', 'str', 'str', 'str', 'str', 'str', 'str', 'str']
- Column 'Date''s format is 'january, 1'.
- User's Question: which team played on december 5?

<Generated Pandas Query>
result = df[df['Date'] == 'december, 5']['Team']

Figure 14: **Insufficient Column Data/Format Information Error by GPT-4.** This error arises when there is a mismatch between the LLM's assumptions about a dataset's structure and the actual data format, leading to incorrect dataframe operations. A notable instance of this occurs when handling date-related queries without clear information on the date format in the dataset. LLM erroneously assumes that the `'Date'` column in the dataset contains full date information including the year. However, in reality, the dataset's `'Date'` column only contains month and day, without the year, leading to a failure in correctly applying the *to_datetime* function. In this case, the LLM's error stems from a gap in understanding the specific format of the `'Date'` column, highlighting the importance of having accurate and sufficient information about the data's structure and format.
Solution: Specifying column formats in prompts, such as stating "`Column 'Date' follows the 'january, 1' format`" effectively addresses Insufficient Column Data/Format Information Errors, ensuring precise LLM data handling.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- User's Question: how does the average shell weight vary across different numbers of rings?

<Generated Pandas Query>
result = df.groupby('Rings').mean()['Shell_weight'] ✗ **CodeLlama-7B**

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- User's Question: how does the average shell weight vary across different numbers of rings?

<Generated Pandas Query>
result = df.groupby('Rings')['Shell_weight'].mean() ✓ **CodeLlama-34B**

Figure 15: **Coding Syntax Error by CodeLlama-7B.** This type of error highlights disparities in the coding capabilities of LLMs, particularly in structuring and executing DataFrame queries. It occurs when the syntax used in a query is incorrect or suboptimal, impacting the query's functionality and efficiency. In the above example, the `.mean()` function is applied across all columns in the grouped DataFrame before selecting the `'Shell_weight'` column. Such an approach is not just inefficient but also potentially problematic. If the DataFrame contains non-numerical columns, computing the mean for all columns initially can lead to errors, as the mean function is not applicable to non-numerical data. This kind of error emphasizes the challenges LLMs face in coding proficiency, particularly regarding the optimization of code for data manipulation tasks.
Solution: Addressing Coding Syntax Errors depends on the LLM's coding expertise. Solutions include choosing a base LLM with enhanced coding abilities, such CodeLlama-34B or GPT-4 or training the LLM on DataFrame QA datasets for better query optimization and data handling skills.

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- User's Question: i'm interested in knowing the most common age of abalone. can you find that for me?

<Generated Pandas Query>
result = df['Age'].mode() ❌

<System Prompt>
You are a professional Python programming assistant. Write Pandas code to get the answer to the user's question.
- Assumptions:
  - The Pandas library has been imported as `pd`. You can reference it directly.
  - The dataframe `df` is loaded and available for use.
  - All string values in the `df` have been converted to lowercase.
- Requirements:
  - Use only Pandas operations for the solution.
  - Store the answer in a variable named `result`.
  - Do NOT include comments or explanations in your code.
  - Place your code between the [PYTHON] and [/PYTHON] tags.

<User Prompt>
You are given a Pandas dataframe named `df`:
- Columns: ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera_weight', 'Shell_weight', 'Rings']
- Data Types: ['str', 'float', 'float', 'float', 'float', 'float', 'float', 'float', 'int']
- The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope.
- User's Question: i'm interested in knowing the most common age of abalone. can you find that for me?

<Generated Pandas Query>
result = df['Rings'].mode() ✔

Figure 16: **Hallucination Error by CodeLlama-7B.** This error type arises when an LLM generates responses based on incorrect assumptions or fabricated details, often due to a lack of domain-specific knowledge. In DataFrame queries, this manifests as references to non-existent columns or data points that the LLM 'hallucinates' or incorrectly infers. In the illstruated example, the accurate approach should involve using the `'Rings'` column, which typically represents the age of abalone. However, a Hallucination Error occurs when the LLM creates a query based on an imaginary `'Age'` column that doesn't exist in the dataset. This error is a result of the LLM's lack of understanding that in the context of abalone, age is commonly denoted by the number of rings, not a separate `'Age'` column. It demonstrates a significant gap in domain-specific knowledge, where the LLM fails to accurately interpret the data context and instead relies on incorrect or made-up information.
Solution: Enhancing prompts with detailed data and column information, like specifying `'Abalone age is assessed by counting rings on the shell'` helps bridge domain knowledge gaps in LLMs, effectively reducing Hallucination Errors and improving query accuracy.