

A Survey on Large Language Model-based Agents for Statistics and Data Science

Maojun Sun^a, Ruijian Han^a, Binyan Jiang^a, Houduo Qi^{a,b},

Defeng Sun^b, Yancheng Yuan^{a*} and Jian Huang^{a,b*}

^aDepartment of Data Science and Artificial Intelligence, The Hong Kong Polytechnic University

^bDepartment of Applied Mathematics, The Hong Kong Polytechnic University

January 14, 2025

Abstract

In recent years, data science agents powered by Large Language Models (LLMs), known as “data agents,” have shown significant potential to transform the traditional data analysis paradigm. This survey provides an overview of the evolution, capabilities, and applications of LLM-based data agents, highlighting their role in simplifying complex data tasks and lowering the entry barrier for users without related expertise. We explore current trends in the design of LLM-based frameworks, detailing essential features such as planning, reasoning, reflection, multi-agent collaboration, user interface, knowledge integration, and system design, which enable agents to address data-centric problems with minimal human intervention. Furthermore, we analyze several case studies to demonstrate the practical applications of various data agents in real-world scenarios. Finally, we identify key challenges and propose future research directions to advance the development of data agents into intelligent statistical analysis software.

Keywords: *data agents; generative AI; data analysis; natural language interaction; statistical software.*

* Corresponding authors.

mj.sun@connect.polyu.hk

{ruijian.han, by.jiang, houduo.qi, defeng.sun, yancheng.yuan, j.huang}@polyu.edu.hk

1 Introduction

As nearly every aspect of society becomes digitized, data analysis has emerged as an indispensable tool across various industries (Inala et al., 2024). For instance, financial institutions leverage data analysis to make informed decisions about stock trends (Provost and Fawcett, 2013; Institute, 2011), hospitals utilize it to monitor patients' health conditions (Waller and Fawcett, 2016), and companies employ it to develop strategic plans (Chen et al., 2012). Despite its widespread utility, data analysis is often perceived as a challenging field with a significant "entry barrier" (Cao, 2017; Jordan and Mitchell, 2015), typically requiring knowledge in areas such as statistics, data science, and computer science (Kitchin, 2014). Since the release of SPSS (IBM, 1968) in 1968, followed by SAS (Inc., 1976), Matlab (MathWorks, 1984), Excel (Microsoft, 1985), Python (Foundation, 1991), R (for Statistical Computing, 1995), PowerBI (Microsoft, 2013), and other specialized data analysis tools and programming languages, these advancements have significantly aided professionals in conducting statistical experiments and data analysis. Moreover, they have made data analysis more accessible to a broader range of practitioners (Witten et al., 2016).

The general data analysis process typically involves several key steps. Initially, data is collected from studies or extracted from databases and imported into tools such as Excel. Next, software like Excel or programming languages such as Python and R are employed to clean and analyze the data, aiming to extract valuable insights. Subsequently, data visualization is performed to make these insights more accessible and understandable. For more complex tasks, such as statistical inference and predictive analysis, statistical and machine learning models are often necessary. This involves data processing, feature engineering, modeling, evaluation, and more. Upon completing the analysis, a final report is usually drafted to summarize the findings and insights. However, for individuals without

expertise in statistics, data science, and programming, data analysis remains a high-barrier task.

The barriers to data analysis primarily exist in the following areas:

- Lack of systematic statistical training: Individuals without a background in statistics may find it challenging to understand which types of analysis are feasible, even when data is presented to them. As data and models become increasingly complex, gaining a solid understanding of current statistical techniques typically requires at least a Master's level of statistical training.
- Software limitation: Simple data analysis tools like Excel are inadequate for complex scenarios, such as predictive analysis or analyzing data from enterprise databases. Conversely, advanced programming languages for data analysis, such as Python and R, require prior programming knowledge, which can be a barrier for many users.
- Challenges in domain-specific problems: In specialized fields like protein or genetic data analysis, general data scientists may find it difficult to perform effective analysis due to a lack of domain-specific knowledge.
- Difficulty in integrating domain knowledge: Corresponding to the last point, domain experts often lack the data science and programming skills needed to quickly incorporate their expertise into data analysis tools. For example, PSAAM (Steffensen et al., 2016) is software designed for the curation and analysis of metabolic models, yet a biologist researching metabolism might find it challenging to integrate this analytical method into common data analysis tools like Excel or R.

With the rise of generative AI, new opportunities have emerged in data science and analysis. This technology is gradually addressing existing challenges while introducing a

new paradigm for approaching data analysis tasks.

This survey explores recent advancements in data agents and showcases data analysis performed by various data agents through several case studies. In Section 2, we briefly discuss the opportunities introduced by recent developments in generative AI. Section 3 reviews and categorizes relevant works on data science agents from 2023. We then present several case studies in Section 4. Section 5 discusses the challenges and future directions in this field, followed by our conclusions in Section 6.

2 Opportunities Brought by Generative AI

The rise and potential of generative AI, particularly large language models (LLMs) or vision language models (VLMs) in the field of data science and analysis have gained increasing recognition in recent years. In addition to understand text, LLMs are also trained to understand tabular data, allowing them to effectively extract insights, identify patterns, and draw meaningful conclusions from tables (Dong and Wang, 2024). Consequently, LLMs have emerged as powerful tools capable of significantly enhancing and transforming a variety of data-driven applications and workflows (Nejjar et al., 2023; Tu et al., 2023; Cheng et al., 2023). Recent research has focused on designing LLM-based data science agents (data agents) to automatically address data science tasks through natural language, as demonstrated by tools like ChatGPT-Advanced Data Analysis (ChatGPT-ADA) (GLM, 2024) and ChatGLM-Data Analysis (ChatGLM-DA) (OpenAI, 2023).

The emergence of data agents offers a potential solution to the previously mentioned challenges, as they lower the entry barrier for users who lack programming or statistical knowledge. By providing an intuitive interface that harnesses the capabilities of LLMs, users can request analyses using natural language, and the data agents can interpret these

instructions, access relevant data, and autonomously apply appropriate analytical techniques. For example, a user might request, “Calculate the sales growth in different regions from 2021 to 2028, generate a bar chart to visualize the results, and provide key insights.” With this simplified instruction, data agents can automatically extract, analyze, visualize, and report data, reducing the requirement for technical expertise and fostering a more efficient workflow. This significantly lowers the entry barriers for individuals unfamiliar with traditional data analysis tools and methods. Figure 1 illustrates the contrast between the traditional paradigm of data analysis tools and the new paradigm introduced by data agents.

Furthermore, by embedding specialized knowledge into LLMs, data agents can potentially overcome challenges faced by data scientists in fields like genomics, where domain expertise is crucial (Cao, 2017). Simultaneously, domain experts who may lack data science or programming skills can rely on data agents to seamlessly integrate their expertise into data analysis workflows. This ability to bridge the gap between domain expertise and data science has the potential to advance interdisciplinary research and decision-making in complex scenarios.

3 LLM-based Data Science Agents

3.1 Overview

LLM-based data agents leverage the powerful natural language understanding and generation capabilities of large language models (LLMs) to autonomously tackle complex data analysis tasks. Figure 2 illustrates a commonly used framework for these agents.

In this framework, the LLM serves as the core of the entire system, driving its performance

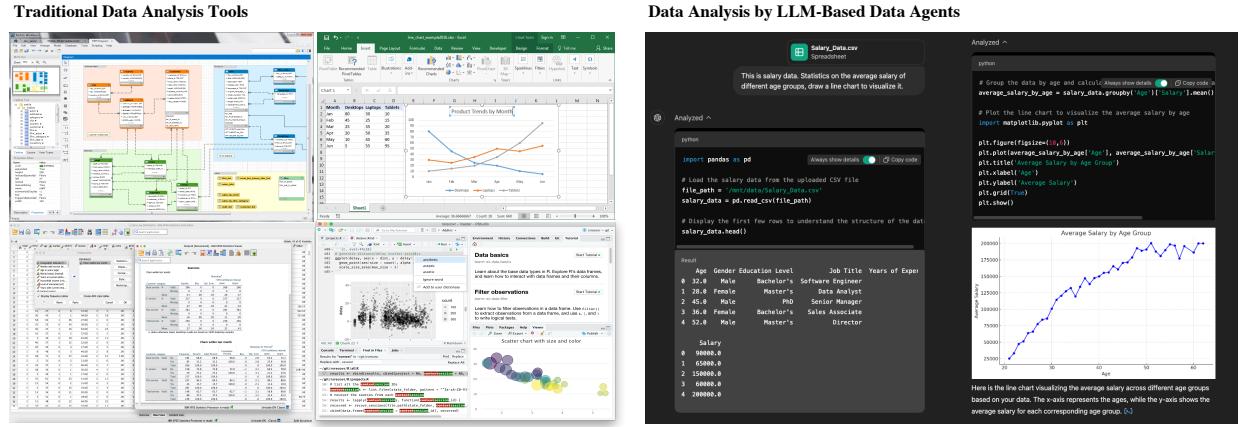


Figure 1: Traditional paradigm of data analysis with tools and new paradigm brought by generative AI. On the left, traditional data analysis tools include MySQL database, Excel, Python, SPSS and R. On the right, data analysis is conducted through natural language interactions with ChatGPT, enabling intuitive, conversational queries and automated data insights, streamlining the analytical process without requiring extensive technical input (Gu et al., 2024).

and reliability. As such, the capabilities of the LLM are critical to the system’s effectiveness, with advanced models like GPT-4 often being used. Data analysis typically involves multiple steps, especially when addressing complex tasks. Techniques such as Planning, Reasoning, and Reflection help ensure that the LLM processes these tasks with greater logical coherence and makes optimal use of its knowledge.

In the architecture, the LLM generates the code for a given data analysis task, executes it, and retrieves the corresponding results. This requires an execution environment, represented by the Sandbox, which safely isolates the code execution process. The Sandbox allows users to run programs and access files without risking the underlying system or platform. It includes pre-installed programming environments and software, such as Python, R, Jupyter, and SQL Server.

A user-friendly interface is also essential to improving usability. An intuitive interface not only attracts users but also enables them to quickly engage with and utilize the system effectively.

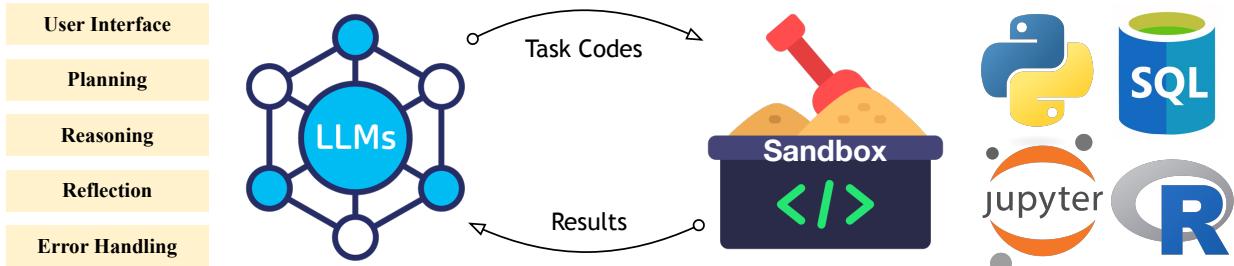


Figure 2: An architecture of an LLM-based data agent. The diagram illustrates the interaction between LLMs and a sandbox environment. On the left, key components of LLMs are highlighted, including User Interface, Planning, Reasoning, Reflection, and Error Handling. The sandbox, positioned centrally, serves as a controlled environment for executing task codes and generating results. On the right, various tools and software that can be pre-installed in the sandbox, such as Python, SQL, Jupyter, and R, indicate the diverse ecosystems where LLM-powered agents can operate.

3.2 Evolution of Data Science Agent

Research on data agents began gaining momentum in 2023. Chandel et al. (2022) trained and evaluated a model within a Jupyter Notebook to predict code based on given commands and results. Soon after, it was discovered that LLMs, such as GPT, could generate accurate code for basic data analysis. With the rise of the LLM-based agent, researchers began designing special data agents for automating data science and analysis tasks by human language. Figure 3 shows some selected works from 2023, while Table 1 illustrates some key characteristics.

3.3 User Interface

The user interface is crucial for attracting users at first glance. Current research on user interface design can be broadly categorized into four types: Integrated Development Environment-based (IDE-based), Independent System, Command line-based (Command-based), and Operation System-based (OS-based).

IDE-based Tools like Jupyter-AI, Chapyter, and MLCopilot integrate LLM-based agents within IDEs such as Jupyter Notebook. This IDE-based approach is highly intuitive and

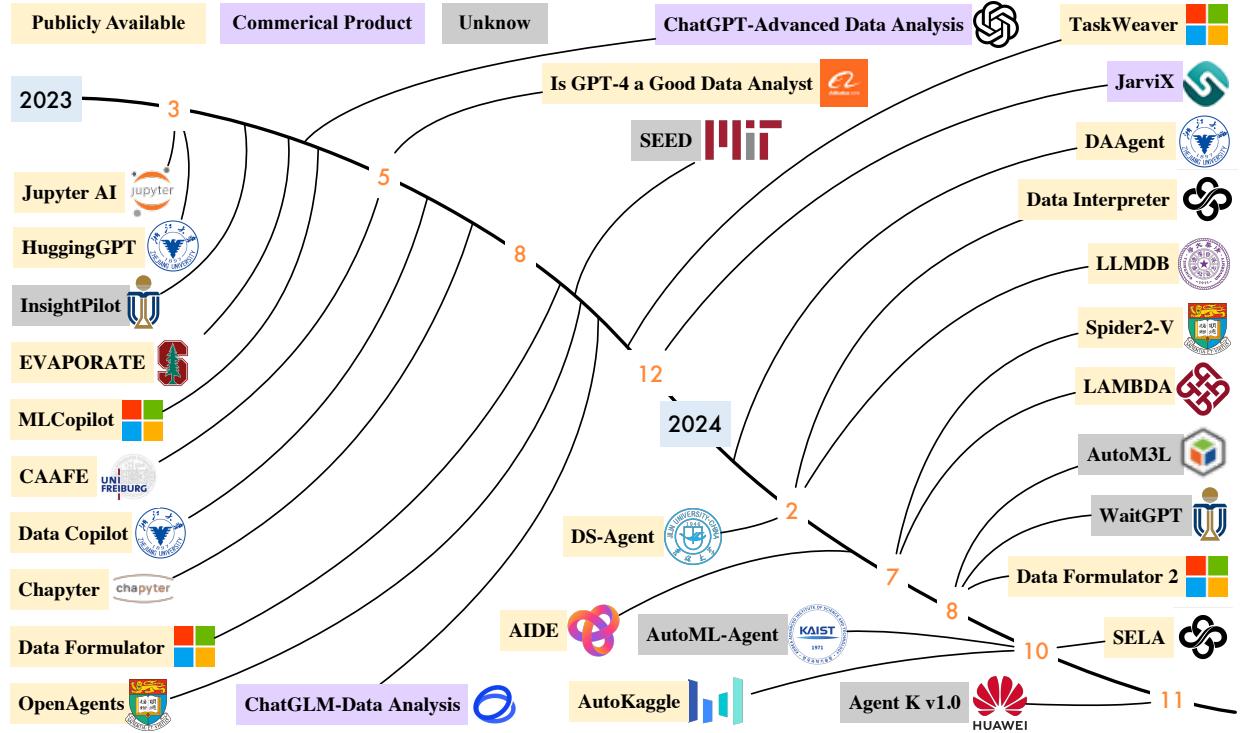


Figure 3: Timeline of selected related works from 2023.

flexible for users who are familiar with the environment. Users can choose to activate the agent within a cell or write and run their own code. Typically, magic commands correspond to specific functions, such as configuring settings or activating the agent. For example, in Jupyter AI, users can initiate the agent by entering the magic command “`%%ai chatgpt`” generate a random matrix with 3×3 ” in a cell. This triggers the agent, designates ChatGPT as the LLM for the current task, and prompts it to generate the code for a 3×3 random matrix. The generated code and its execution results are then displayed as output below the cell. This interface is particularly popular among users with programming experience, as it provides significant flexibility.

Independent System Some works have focused on developing independent systems equipped with user interfaces. For example, ChatGPT introduced a streamlined, intuitive conversational system—a model of interaction that has been widely adopted in subsequent projects. In the context of data analysis tasks, beyond basic text-based input and output,

Table 1: Characteristics of selected data agents. Methods can be categorized into Conversational and End-to-End approaches. Conversational methods are typically designed for interactive dialogue, allowing users to guide the process through multiple prompts and feedback. In contrast, End-to-End approaches involve a single prompt, where the agent autonomously generates a plan and solves the problem sequentially. The user interface can be categorized into four types: IDE-based, independent systems (Systems), command line interface (CLI), and operating-system-based. The term “Human-in-the-Loop” indicates that humans can intervene in the data agent’s workflow, such as modifying code in situations where automatic processes are inadequate. “Self-Correcting” refers to the agent’s ability to automatically identify and correct errors within the workflow through reflection. Finally, “Expandable” denotes the data agent’s capacity to incorporate customized tools or knowledge. “-” indicates that the attribute is either not mentioned in the paper or could not be observed from the provided resources.

Data Agents	Methods	User Interface	Planning	Human in the Loop	Self-correcting	Expandable
ChatGPT-ADA (OpenAI, 2023)	Conversational	System	Linear	✗	✓	✗
Data Copilot (Zhang et al., 2023b)	End-to-end	System	Linear	✗	✓	✗
Jupyter AI (jupyterlab, 2023)	Conversational	IDE-based	Basic IO	✓	✗	✗
MLCopilot (Zhang et al., 2023a)	Conversational	IDE-based	Basic IO	✓	✗	✗
Chapyter (chapyter, 2023)	Conversational	IDE-based	Basic IO	✓	✗	✗
Openagents (Xie et al., 2023)	Conversational	System	Linear	✗	✗	✓
JarviX (Chen et al., 2024)	End-to-end	-	-	-	-	-
DS-Agent (Guo et al., 2024)	End-to-end	CLI	Linear	✗	✓	-
Spider2-V (Cao et al., 2024)	End-to-end	OS-Based	-	✗	✓	-
ChatGLM-DA (GLM, 2024)	Conversational	System	Linear	✗	✓	✗
TaskWeaver (Qiao et al., 2023)	End-to-end	CLI & System	Linear	✗	✓	✓
Data Interpreter (Hong et al., 2024)	End-to-end	CLI	Hierarchical	✓	✓	✓
LAMBDA (Sun et al., 2024)	Conversational	System	Basic IO	✓	✓	✓
Data Formulator 2 (Wang et al., 2024a)	Conversational	System	Basic IO	✗	✓	-
AutoM3L (Luo et al., 2024)	End-to-end	-	-	✗	-	✓
SELA (Chi et al., 2024)	End-to-end	CLI	Hierarchical	✗	✓	-
AIDE (Jiang et al., 2024)	End-to-end	CLI	Hierarchical	✗	✓	-
AutoKagle (Li et al., 2024)	End-to-end	CLI	Linear	✓	✓	✓
AutoML-Agent (Trirat et al., 2024)	End-to-end	-	Linear	-	✓	-
Agent K v1.0 (Grotnit et al., 2024)	End-to-end	-	Hierarchical	-	✓	✓

several systems have introduced specialized features, such as visualization, report generation, and file download options, to simplify user interactions. For instance, LAMBDA (Sun et al., 2024) facilitates easy data review by enabling intuitive data display after users upload their data. Data Formulator 2 (Wang et al., 2024a) further enhances the iterative process of creating data visualizations through a multimodal interface, combining graphical user interface (GUI) elements with natural language inputs, allowing users to specify their visualization intentions with both precision and flexibility. WaitGPT (Xie et al., 2024) addresses the challenge of understanding and verifying LLM-generated code by transforming

raw code into an interactive, step-by-step visual representation. This allows users to comprehend, validate, and adjust specific data operations, actively guiding and refining the analysis process.

Command-based Works like Data Interpreter (Hong et al., 2024) and TaskWeaver (Qiao et al., 2023) using command-line interfaces (CLI) in their works. For researchers and experienced users, it provides greater flexibility and control over the system, allowing users to execute a wide range of functions in the command line and customize their actions. Besides, command-based interfaces often require less computational overhead compared to graphical user interfaces, making them more efficient.

OS-based OS-based agents, such as UFO (Zhang et al., 2024), are designed to operate directly within an operating system environment, allowing them to control a wide range of system tasks and resources. Similarly, Spider2-V (Cao et al., 2024) simulates the typical workflow of a data scientist by mimicking actions such as clicking, typing, and writing code, providing an OS-level interactive experience that closely resembles how humans manage data science tasks. However, while OS-based agents like Spider2-V lay a solid foundation for user interaction, achieving full automation of the data science workflow remains an ongoing challenge (Cao et al., 2024).

3.4 Planning, Reasoning, and Reflection

Planning, Reasoning, and Reflection often play crucial roles in guiding the actions of data agents. Planning, in particular, focuses on generating a logically structured roadmap of actions or thought processes for solving specific problems (Huang et al., 2024; Hong et al., 2024). When a data agent receives a request, it typically needs to provide a response along with execution results. Complex tasks often require a step-by-step approach to ensure

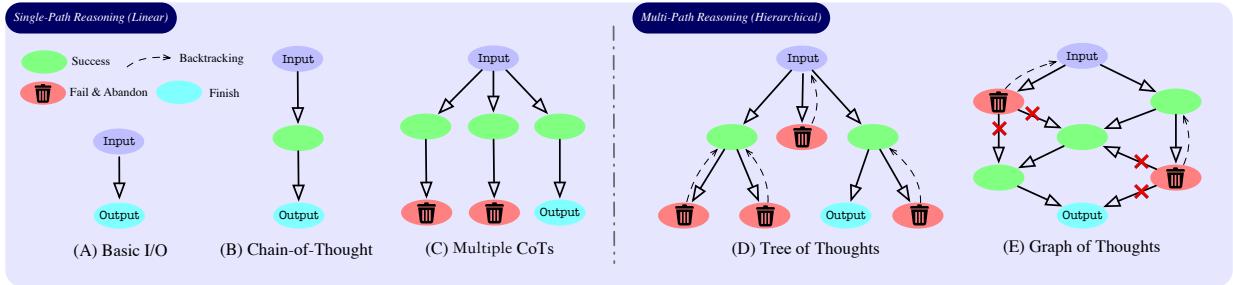


Figure 4: Commonly used planning and reasoning strategies in LLM-based data agents for organizing tasks or solving problems. Each node represents a sub-task in the roadmap.

effective resolution, while simpler tasks can be handled without such detailed breakdowns.

Some approaches focus on building conversational data agents (Zhang et al., 2023b,a; Sun et al., 2024), where users interact with the agent over multiple rounds to complete a task. In these cases, under human supervision, complex planning is not necessary, as guidance can simplify decision-making and adjust the workflow dynamically. Some of these works operate in a Basic I/O mode. On the other hand, End-to-end data agents (Guo et al., 2024; Qiao et al., 2023; Hong et al., 2024; Chi et al., 2024; Jiang et al., 2024; Li et al., 2024; Trirat et al., 2024; Grosnit et al., 2024) are designed to allow users to issue a single prompt that encompasses all requirements. In these cases, the agent employs planning, reasoning, and reflection to iteratively complete all tasks autonomously.

Recent research in planning has introduced two main approaches: Linear Structure Planning (or Single Path Planning/Reasoning) and Hierarchical Structure Planning (or Multiple Path Planning/Reasoning). Figure 4 illustrates these different planning techniques. A task can be planned and broken down into sub-tasks, as demonstrated in methodologies like Chain-of-Thought (CoT) (Wei et al., 2022), ReAct (Yao et al., 2022), Tree-of-Thoughts (ToT) (Yao et al., 2024), and Graph-of-Thoughts (GoT) (Besta et al., 2024).

Linear Structure Planning In linear structure planning, a task is decomposed into a sequential, step-by-step process. For example, DS-Agent (Guo et al., 2024) utilizes

Case-Based Reasoning to retrieve and adapt relevant insights from a knowledge base of past successful Kaggle solutions. This enables the system to develop experiment plans, iteratively adjusting the plan based on execution feedback. This approach allows the agent to learn from previous experiences and continuously improve its performance. Similarly, the Planner agent in AutoKaggle (Li et al., 2024) is responsible for task decomposition and plan generation, using logical reasoning to determine the sequence of actions for other agents to execute. AutoML-Agent (Trirat et al., 2024) adopts a retrieval-augmented planning (RAP) strategy to generate diverse plans for AutoML tasks. By leveraging the knowledge embedded in LLMs, information retrieved from external APIs, and user requirements, RAP allows the agent to explore a wider range of potential solutions, leading to more optimal plans.

Hierarchical Structure Planning Simple linear planning is often insufficient for complex tasks. Such tasks may require hierarchical and dynamic, adaptable plans that can account for unexpected issues or errors in execution (Hong et al., 2024). For instance, Hong et al. (2024) utilizes a hierarchical graph modeling approach that breaks down intricate data science problems into manageable sub-problems, represented as nodes in a graph, with their dependencies as edges. This structured representation enables dynamic task management and allows for real-time adjustments to evolving data and requirements. Additionally, they further introduce “Programmable Node Generation,” to automate the generation, refinement, and verification of nodes within the graph, ensuring accurate and robust code generation. AIDE (Jiang et al., 2024) employs Solution Space Tree Search to iteratively improve solutions through generation, evaluation, and selection components. Similarly, SELA (Chi et al., 2024) combines LLMs with Monte Carlo Tree Search (MCTS) to enhance AutoML performance. It starts by using LLMs to generate insights for various machine learning

stages, creating a search space for solutions. MCTS then explores this space by iteratively selecting, simulating, and back-propagating feedback, enabling the discovery of optimal pipelines. Agent K v1.0 (Grognit et al., 2024) employs a structured reasoning framework with memory modules, operating through multiple phases. The first phase, automation, handles data preparation and task setup, generating actions through structured reasoning. The second phase, optimization, involves solving tasks and enhancing performance using techniques such as Late-Fusion Model Generation and Bayesian optimization. The final phase, generalization, utilizes a memory-driven system for adaptive task selection.

Reflection Reflection enables an agent to evaluate past actions and decisions, adjust strategies, and improve future task performance. This process is essential for self-correction and debugging during task execution. For example, Wang et al. (2024b) employs trajectory filtering to train agents that can learn from interactions and enhance their self-debugging capabilities. This technique involves selecting trajectories in which the model initially makes errors but successfully corrects them through self-reflection in subsequent interactions. Similarly, Zhang et al. (2023b) and Sun et al. (2024) use self-reflection based on code execution feedback to address errors. If a compilation error occurs, the agents repeatedly attempt to revise the code until it runs successfully or a maximum retry limit is reached. This iterative process helps ensure code correctness and usability.

3.5 Multi-agent Collaboration

Multi-agent collaboration involves multiple agents working together to achieve common goals. In this setup, agents communicate, negotiate, and share information to optimize their collective performance (Xi et al., 2023). In the context of data agents, dividing complex tasks among agents with specialized expertise can enhance both the efficiency and

performance (Sun et al., 2024). For example, LAMBDA operates with two distinct agents: the Programmer and the Inspector. The Programmer is responsible for generating code based on user instructions. Meanwhile, the Inspector plays a critical role in debugging the generated code as needed, ensuring both functionality and accuracy in the execution (Sun et al., 2024). AutoML-Agent (Trirat et al., 2024) involves multiple agents—such as the Agent Manager, Prompt Agent, Operation Agent, Data Agent, and Model Agent—that together cover the entire pipeline, from data retrieval to model deployment. Similarly, AutoKaggle (Li et al., 2024) employs five specialized agents—Reader, Planner, Developer, Reviewer, and Summarizer—to manage each phase of the process, ensuring comprehensive analysis, effective planning, coding, quality assurance, and detailed reporting. OpenAgents (Xie et al., 2023) also adopts a multi-agent approach, with agents such as the Data Agent, Plugins Agent, and Web Agent collaborating to solve a variety of tasks.

3.6 Knowledge Integration

Integrating domain-specific knowledge into data agents presents a challenge (Dash et al., 2022; Sun et al., 2024). For example, when a domain expert has specialized knowledge, such as specific protein analysis code, the agent system are expected able to incorporate and apply this knowledge effectively. One approach is tool-based, where the expert’s analysis code is treated as a tool that is recognizable by the LLM (Xie et al., 2023). When the agent encounters a relevant problem, it can call upon the appropriate tool from its library to execute the specialized analysis. Another method involves the Retrieval-Augmented Generation (RAG) technique (Lewis et al., 2020), where relevant code is first retrieved and then embedded within the context to facilitate in-context learning. LLM-based agents can also access and interact with external knowledge sources, such as databases or knowledge

graphs, to augment their reasoning capabilities (Wang et al., 2024b).

Sun et al. (2024) proposes a Knowledge Integration method that builds on this concept. In LAMBDA, analysis codes are parsed into two parts: descriptions and executable code. These are then stored in a knowledge base. When the agent receives a task, it retrieves the relevant knowledge based on the similarity between the task description and the descriptions stored in the knowledge base. The corresponding code is then used for in-context learning (ICL) or back-end execution, depending on the configuration. This approach enables agents to effectively leverage domain-specific knowledge in relevant scenarios.

3.7 System Design and Other Related Works

Recent advancements in interactive data science systems highlight a variety of approaches in system design, with LLMs and structured frameworks significantly enhancing the user experience across key areas such as data visualization, task specification, predictive modeling, and data exploration. Notable systems like VIDS (Hassan et al., 2023), Data-Copilot (Zhang et al., 2023b), InsightPilot (Ma et al., 2023), and JarviX (Liu et al., 2023) exemplify diverse design principles tailored to these specific functions. For instance, VIDS employs a four-stage dialogue model, featuring “micro-agents” dedicated to tasks such as data visualization, task formulation, prediction engineering, and results summarization. This structure optimizes system responsiveness at each stage (Hassan et al., 2023). Data-Copilot adopts a code-centric approach, generating intermediate code to process data and subsequently transforming it into visual outputs, such as charts, tables, and summaries (Zhang et al., 2023b).

Other frameworks emphasize workflow automation. InsightPilot integrates an “insight engine” that guides data exploration, reducing LLM hallucinations and enhancing the accuracy of exploratory tasks (Ma et al., 2023). JarviX, in combination with MLCopilot (Zhang

et al., 2023a), contributes to automated machine learning by merging LLM-driven insights with AutoML pipelines. JarviX, in particular, offers a no-code interface, democratizing complex data analysis and streamlining predictive modeling processes (Liu et al., 2023). Additionally, in the domain of database management, systems like LLMDB improve efficiency and reduce hallucinations and computational costs during tasks such as query rewriting, database diagnosis, and data analytics. This is achieved by embedding domain-specific knowledge, semantic search, caching, and multi-round inference techniques (Zhou et al., 2024).

In data visualization, MatPlotAgent (Yang et al., 2024) transforms raw data into clear, informative visualizations by leveraging both code-based and multimodal LLMs. Collectively, these systems demonstrate success in designing sub-modules and processes within the broader data analysis workflow.

Moreover, Wang et al. (2024a) organizes user interactions into "data threads" to provide context and facilitate the exploration and revision of prior steps. A similar approach is seen in Xie et al. (2024), which transforms raw code into an interactive visual representation. This provides a step-by-step visualization of LLM-generated code in real-time, allowing users to understand, verify, and modify individual data operations. Such an interface empowers users to guide and refine the analysis process proactively, facilitating better monitoring and steering of data analysis performed by LLMs. SEED (Chen et al., 2024) combines LLMs with methods like code generation and small models to produce domain-specific data curation solutions. HuggingGPT (Shen et al., 2024), on the other hand, uses LLMs to coordinate a variety of expert models from platforms such as Hugging Face, solving a broader range of AI tasks across multiple modalities.

4 Data Analysis Through Natural Language Interaction: Case Studies

In this section, we present a series of case studies conducted by a diverse range of agents, each illustrating the new data analysis paradigm facilitated through natural language interaction. These case studies demonstrate how this approach enables users to engage with data more intuitively and effectively, breaking down traditional barriers to data accessibility and understanding. By leveraging natural language processing, these agents can interpret and respond to complex queries, providing insights that are both comprehensive and easily digestible. Through these examples, we aim to highlight the transformative potential of natural language interaction in data analysis.

4.1 Case study 1: Data Visualization and Machine Learning by Conversational Data Agents

In this case study, we utilized ChatGPT and LAMBDA to demonstrate data visualization and machine learning, respectively. Specifically, we first used ChatGPT to explore the effect of alcohol content on the quality of different types of wine, focusing on both red and white varieties. Then, we used LAMBDA for machine learning and automatically generate reports.

We utilized the Wine Quality dataset¹ and employed ChatGPT-Advanced Data Analysis to conduct exploratory data analysis and visualize the influence of alcohol content on wine quality ratings. Figure 5 illustrates the detailed planning and problem-solving process.

GPT-4 first analyzed the problems and then outlined a step-by-step plan to solve the

¹<https://archive.ics.uci.edu/dataset/186/wine+quality>

(1)
Let's begin with loading and examining the dataset, then proceed with the data preparation and visualization steps as you've outlined.

I'll:

- 1 Load the data and inspect for any missing values.
- 2 Separate the data by wine type (red and white) if this information is available.
- 3 Visualize the distribution of wine quality scores.
- 4 Plot a box plot to show the relationship between alcohol content and quality.

I'll start by loading the data and checking its structure.

```
import pandas as pd

# Load the uploaded dataset
file_path = '/mnt/data/wine_quality.csv'
wine_data = pd.read_csv(file_path)

# Display basic information about the dataset, including
# checking for missing values
wine_data.info = wine_data.info()
wine_data_head = wine_data.head()
missing_values = wine_data.isnull().sum()

wine_data_info, wine_data_head, missing_values

Result:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496 Data columns
(total 13 columns): # Column Non-Null Count
Dtype: --- fixed_acidity
6497 non-null float64 1 volatile_acidity 6497 non-
null float64 2 citric_acid 6497 non-null float64 3
residual_sugar 6497 non-null float64 4 chlorides
6497 non-null float64 5 free_sulfur_dioxide 6497
non-null float64 6 total_sulfur_dioxide 6497 non-
null float64 7 density .....  

.....
```

(None, fixed_acidity volatile_acidity citric_acid residual_sugar chlorides \ 0 7.4 0.70 0.00 1.9 0.076 1 7.8 0.88 0.00 2.6 0.098 2 7.8 0.76 0.04 2.3 0.092 3 11.2 0.28 0.56 1.9 0.075 4 7.4 0.70 0.00 1.9 0.076

The dataset has 6497 entries and 13 columns, with no missing values. It includes a color column, which identifies wine type (red or white).

(2)
Here's the plan for the next steps:

- 1 **Separate the data** by wine type to explore red and white wines independently.
- 2 **Visualize the wine quality distribution** with a histogram.
- 3 **Create a box plot** of alcohol content across different quality scores.

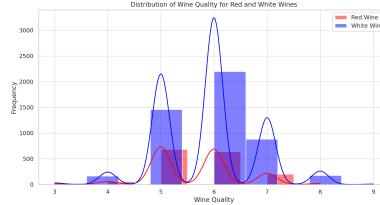
I'll proceed with these visualizations now.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Set the style for the plots
sns.set(style="whitegrid")
```

```
# Separate data by wine type
red_wine = wine_data[wine_data['color'] == 'red']
white_wine = wine_data[wine_data['color'] == 'white']

# Plot the distribution of wine quality for both types
plt.figure(figsize=(12, 6))
sns.histplot(red_wine['quality'], bins=10, color='red',
label='Red Wine', kde=True)
sns.histplot(white_wine['quality'], bins=10, color='blue',
label='White Wine', kde=True)
plt.title('Distribution of Wine Quality for Red and White Wines')
plt.xlabel('Wine Quality')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

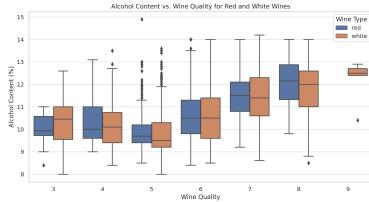


This histogram shows the distribution of wine quality scores for both red and white wines, with a slight skew toward the middle scores in both categories.

(3)

Next, I'll create a box plot to compare the alcohol content across different quality scores.

```
# Create a box plot for alcohol content across wine
# quality scores
plt.figure(figsize=(12, 6))
sns.boxplot(data=wine_data, x='quality', y='alcohol',
hue='color')
plt.title('Alcohol Content vs. Wine Quality for Red and White Wines')
plt.xlabel('Wine Quality')
plt.ylabel('Alcohol Content (%)')
plt.legend(title='Wine Type')
plt.show()
```



The box plot illustrates how alcohol content varies across different quality scores for red and white wines. Generally, higher quality scores are associated with slightly higher alcohol content, especially in white wines. This offers insight into the relationship between alcohol levels and perceived quality.

Would you like to explore additional comparisons or analyses based on other chemical properties?

Figure 5: Response of ChatGPT-Advanced Data Analysis in Case Study 1. The numbers above represent the work done by ChatGPT in each step.

tasks. The entire workflow proceeded smoothly, with the code running efficiently to load the data, check for missing values, and generate visualizations, with each step delivering accurate results. Its ability to interpret data and provide insights significantly streamlined the analytical process. Finally, it provided insights into the relationship between quality scores and alcohol content.

However, for some commercial products, such as ChatGPT and ChatGLM, there may be restrictions when attempting to install additional packages. Due to the server's protection mechanisms, the sandbox environment does not have internet access, and the agent receives a notification stating, "There is no internet access." Any attempt to install packages results

ChatGPT-Advanced Data Analysis	ChatGLM-Data Analysis
<pre># Let's use the provided example and adapt it to set the minSup to 500. from PAMI.frequentPattern.basic import FPGrowth as alg # Define the file URL and minimum support fileURL = "https://u-aizu.ac.jp/~udayrage/datasets/transactionalDatabases/ Transactional_T10I4D100K.csv" minSup = 500 # Create an instance of the FPGrowth algorithm with the specified parameters obj = alg.FPGrowth(iFile=fileURL, minSup=minSup, sep="\t") # Start the mining process obj.mine() # Get the frequent patterns as a DataFrame frequentPatternsDF = obj.getPatternsAsDataFrame() # Print the total number of patterns total_patterns = len(frequentPatternsDF) total_patterns</pre> <hr/> <pre>ModuleNotFoundError Traceback (most recent call last) Cell In[2], line 2 1 # Let's use the provided example and adapt it to set the minSup to 500. ----> 2 from PAMI.frequentPattern.basic import FPGrowth as alg 4 # Define the file URL and minimum support 5 fileURL = "https://u-aizu.ac.jp/~udayrage/ datasets/transactionalDatabases/Transactional_T10I4D100K.csv" ModuleNotFoundError: No module named 'PAMI' ✘</pre> <p>It seems that the PAMI package is not installed in this environment, and without internet access, I can't install it directly.</p>	<p>You aim to use the PAMI library to perform pattern mining on a transactional database. The dataset you provided is: 'https://u-aizu.ac.jp/~udayrage/datasets/transactionalDatabases/Transactional_T10I4D100K.csv'.</p> <p>Set the minimum support (minSup) to 500. Use the FP Growth algorithm for mining. Save the frequent patterns mined to a file. Retrieve the patterns as a DataFrame and display the total count of patterns, runtime, and memory usage details.</p> <p>Execution Plan:</p> <p>First, I'll ensure the PAMI library is installed. Then, I'll download the dataset and use the FP Growth algorithm for pattern mining. Next, I'll execute code to save the frequent patterns, retrieve the patterns as a DataFrame, and display the requested details</p> <pre>!pip install pam</pre> <p>This response has been terminated. ✘</p>

Figure 6: Scenarios when the task needs to install additional packages. The protection mechanism will prevent the installation and further terminate the task. Illustrated by ChatGPT-ADA and ChatGLM-DA.

in an error. Figure 6 illustrates this issue.

Next, we aim to train a set of models to predict wine quality using LAMBDA. LAMBDA facilitates an interactive machine learning process, enabling us to perform tasks such as data processing, feature engineering, model training, parameter tuning, and evaluation through a series of guided conversations. Finally, we leveraged LAMBDA’s built-in report generation feature to compile a comprehensive analysis, which includes details of the tasks completed in the conversation history. The case, including the conversation and the generated report, is presented in Figure 7.

As beginner-level users, we first asked LAMBDA to recommend some models, and it suggested advanced options like XGBoost. Next, we tasked LAMBDA with basic data preprocessing, which it handled correctly. We then trained and evaluated the recommended models using 5-fold cross-validation, a task LAMBDA performed exceptionally well, even

Question Code **Execution Result** **Answer**

(Q) This is the wine quality dataset. I want to do a regression task, can you recommend some models for me? (1)

(A) Certainly! For a regression task on the wine quality dataset, you can consider the following models:

1. **Linear Regression**: A simple and interpretable model that assumes a linear relationship between the features and the target variable.
2. **Random Forest Regressor**: An ensemble method that builds multiple decision trees and averages their predictions, which often provides good performance.
3. **Gradient Boosting Regressor** (e.g., XGBoost, LightGBM, or CatBoost)**: These models sequentially build decision trees to correct errors made by previous trees, often leading to high accuracy.
4. **Support Vector Regression (SVR)**: A model that uses support vectors to find the best fit line in a high-dimensional space.
5. **K-Nearest Neighbors (KNN) Regressor**: A non-parametric model that predicts the target variable based on the average of the nearest neighbors.

You can start with Linear Regression for simplicity and interpretability, and then move on to more complex models like Random Forest or Gradient Boosting for potentially better performance.

Would you like to proceed with training one of these models?

(Q) Based on the processed data in the last step, perform a 5-fold Cross-validation on Linear Regression, the evaluation metric is MSE. Print the score and average score. (3)

(A) from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

```
# Initialize the Linear Regression model
model = LinearRegression()

# Perform 5-fold cross-validation and calculate MSE
mse_scores = -cross_val_score(model, X_processed, y, cv=5, scoring='neg_mean_squared_error')

# Print the individual MSE scores and the average MSE score
print("Individual MSE scores:", mse_scores)
print("Average MSE score:", mse_scores.mean())
.....
```

Execution result:
Individual MSE scores: [0.50691704 0.64197325 0.59655807 0.55984441 0.51144414]
Average MSE score: 0.5633473821839552
Individual MSE scores / Average MSE score |

0.50691704		0.5633473821839552
0.64197325		
0.59655807		
0.55984441		
0.51144414		

The 5-fold cross-validation results for the Linear Regression model show an average Mean Squared Error (MSE) of approximately 0.563. This indicates that, on average, the model's predictions are off by about 0.563 units squared. Next, you can compare this performance with other regression models like Random Forest or Gradient Boosting to see if they offer better accuracy.
`linear_regression_model.pkl`

(Q) Train and select some parameters for Support Vector Regression. Print the best parameters and MSE based on cross-validation. (5)

(A) from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV
Define the parameter grid for SVR
param_grid = {
 'C': [0.1, 1, 10],
 'epsilon': [0.01, 0.1, 0.2],
 'kernel': ['linear', 'rbf']
}

Initialize GridSearchCV with 5-fold cross-validation and MSE as the scoring metric
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)

Fit the model to the data
grid_search.fit(X_processed, y)

Best parameters found | Best MSE score |
| [C: 1, 'epsilon': 0.2, 'kernel': 'rbf'] | 0.5240376272769057 |

The Support Vector Regression (SVR) model with the best parameters found during grid search shows an average Mean Squared Error (MSE) of approximately 0.524. This indicates that, on average, the model's predictions are off by about 0.524 units squared. The SVR model outperforms both the Linear Regression, Random Forest, and XGBoost models in terms of MSE.
`best_svr_model.pkl`

(Q) First, for column color, perform one-hot encoding. For all other columns including the target column, perform standardization. (2)

(A) import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
Load the dataset
data = pd.read_csv('Users/xxx/LAMBDA/cache/xxxxxx/wine_quality.csv')
Perform one-hot encoding on the 'color' column
data = pd.get_dummies(data, columns=['color'], drop_first=False)
.....

```
# Separate features and target
X = data.drop('quality', axis=1)
y = data['quality']
# Standardize all features except the one-hot encoded 'color' columns
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X.drop(columns=['color_red', 'color_white']))
# Combine the standardized features with the one-hot encoded 'color' columns
X_processed = pd.concat([pd.DataFrame(X_scaled, columns=X.columns[-2]), X[['color_red', 'color_white']]], axis=1)
# Display the first few rows of the processed data
X_processed.head(5)
```

Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol', 'color_red', 'color_white'], object=True)

0.142473	2.18863	-2.10283	-0.744778	0.056985	-1.10014	-1.44636	1.03499	1.13109	0.193907	-0.91564	True	False
0.451038	2.35324	-2.10283	-0.695754	0.056985	-1.02427	-1.20146	1.11307	1.13109	0.193907	-0.91564	True	False
0.451038	2.35324	-2.10283	-0.666699	0.056985	-0.92427	-1.09242	1.11307	1.13109	0.193907	-0.91564	True	False
0.370856	-2.362438	1.66109	-0.744778	0.341412	-0.862074	-0.905324	1.10169	-0.936363	0.3275	-0.580661	True	False
0.451038	2.35324	-2.10283	-0.744778	0.056985	-1.02427	-1.20146	1.11307	1.13109	0.193907	-0.91564	True	False

The data has been successfully preprocessed with one-hot encoding for the 'color' column and standardization for the numerical features. The next steps could involve splitting the data into training and testing sets, followed by training a regression model such as Linear Regression, Random Forest, or Gradient Boosting to predict the wine quality.

(Q) Train and select some parameters for XGBoost. Print the best parameters and MSE based on cross-validation. (4)

(A) import xgboost as xgb
from sklearn.model_selection import GridSearchCV
Define the parameter grid for XGBoost
param_grid = {
 'n_estimators': [100, 200, 300],
 'max_depth': [3, 5, 7],
 'learning_rate': [0.01, 0.1, 0.2]
}

Initialize GridSearchCV with 5-fold cross-validation and MSE as the scoring metric
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, scoring='.....')

Fit the model to the data
grid_search.fit(X_processed, y)

Best parameters found | Best MSE score |
| [n_estimators: 300, 'max_depth': 5, 'learning_rate': 0.01] | 0.5376976446782151 |

The XGBoost model with the best parameters found during grid search shows an average Mean Squared Error (MSE) of approximately 0.538. This indicates that, on average, the model's predictions are off by about 0.538 units squared. The XGBoost model outperforms both the Linear Regression and Random Forest models in terms of MSE.
`best_xgbosst_model.pkl`

(Q) Train and select some parameters for the KNN Regressor. Print the best parameters and MSE based on cross-validation. (6)

(A) from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV
Define the parameter grid for KNN Regressor
param_grid = {
 'n_neighbors': [3, 5, 7, 9],
 'weights': ['uniform', 'distance'],
 'p': [1, 2] # 1 for manhattan_distance, 2 for euclidean_distance
}

Initialize GridSearchCV with 5-fold cross-validation and MSE as the scoring metric
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)

Fit the model to the data
grid_search.fit(X_processed, y)

Best parameters found | Best MSE score |
| [n_neighbors: 9, 'p': 1, 'weights': 'distance'] | 0.5887054666843026 |

The K-Nearest Neighbors (KNN) Regressor model with the best parameters found during grid search shows an average Mean Squared Error (MSE) of approximately 0.589. This indicates that, on average, the model's predictions are off by about 0.589 units squared. The KNN model performs worse than the SVR, XGBoost, and Random Forest models in terms of MSE.
`best_knn_model.pkl`

Report

3.2 Data Processing:
To prepare the data for modeling, several preprocessing steps were undertaken:

- **One-Hot Encoding:** The 'color' feature was converted to binary columns to indicate red or white wine.
- **Standardization:** Numerical features were standardized using StandardScaler to ensure comparability and improve the model's convergence.

Here is a preview of the preprocessed data:

fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	dens
0.142473	2.18863	-2.10283	-0.744778	0.056985	-1.10014	-1.44636	1.03499
0.451038	2.35324	-2.10283	-0.695754	0.056985	-1.02427	-1.20146	1.11307
0.451038	2.35324	-2.10283	-0.666699	0.056985	-0.92427	-1.09242	1.11307
0.370856	-2.362438	1.66109	-0.744778	0.341412	-0.862074	-0.905324	1.10169
0.451038	2.35324	-2.10283	-0.744778	0.056985	-1.02427	-1.20146	1.11307

3.3 Modeling:
Various regression models were trained on the preprocessed dataset, using cross-validation for evaluation. The models include:

- **Linear Regression:** A fundamental machine learning algorithm that models the relationship between those features and the target variable as a linear equation.
- **Random Forest Regressor:** An ensemble learning method that operates by constructing multiple decision trees and aggregating their results to classify new data points. It emphasizes the importance of model selection and hyperparameter tuning.
- **Support Vector Regression (SVR):** This method uses the concept of support vectors to define a hyperplane that best fits the data points while allowing a margin of error.

4. Results:
The results of model evaluation are summarized in the table below, showcasing the performance of each model with respect to Mean Squared Error (MSE).

Model	Best Parameters	Average MSE Score
Linear Regression	(Default)	0.553
Random Forest	(Default parameters)	0.554
Support Vector Regression (SVR)	C: 1, epsilon: 0.01, max_depth: 5, n_estimators: 300	0.557
XGBoost	learning_rate: 0.01, max_depth: 5, n_estimators: 300	0.557
K-Nearest Neighbors	{'n_neighbors': 9, 'p': 1, 'weights': 'distance'}	0.589

5. Conclusion:
The report details the regression analysis conducted on the Wine Quality dataset using several machine learning models. The results show that the XGBoost model provided the best performance, achieving the lowest Mean Squared Error (MSE). The Random Forest model also performed well, closely matching the XGBoost results. The Support Vector Regression model provided competitive results, demonstrating its effectiveness for regression tasks in this context. The results emphasize the importance of model selection and hyperparameter tuning. Overall, the analysis highlights the strengths and weaknesses of different regression models and provides insights into how machine learning engineering can be applied to achieve high prediction accuracy while also considering ensemble methods for improved robustness.

Figure 7: Conversational machine learning and report generation by LAMBDA.

providing download links for the resulting models. Finally, we used LAMBDA’s report generation feature to create a structured and comprehensive report that effectively captured the key insights.

These cases demonstrate the effectiveness of conversational data agents like ChatGPT and LAMBDA in streamlining the data visualization and machine learning workflow, particularly for users without programming experience.

4.2 Case Study 2: Data Visualization and Machine Learning by End-to-End Data Agents

End-to-end data science agents are particularly convenient for users who wish to perform data-related tasks with a single prompt. In this case study, we demonstrate how an end-to-end data agent, the Data Interpreter, can handle both data visualization and machine learning tasks. Specifically, we prompt the Data Interpreter to visualize the average salary across different age groups using the Salary Data². Figure 8 shows both the prompt and the detailed internal process of the Data Interpreter in executing this task.

We pre-downloaded the data to disk due to errors that may occur during the download process. The Data Interpreter began by generating a path in the Planning phase, which consisted of three task nodes: (1) “Load the salary data from the CSV file,” (2) “Analyze the average salary by different age groups,” and (3) “Draw a line chart to visualize the average salary by age groups.” It then sequentially executed each task. After obtaining the results for each step, the interpreter checked whether the current task was successfully completed. If not, it re-executed the task. This reflection process ensures task completion and accuracy by allowing for iterative re-evaluation. Once all nodes are successfully completed, the overall

²<https://www.kaggle.com/datasets/mohithsairamreddy/salary-data>

(Q): I want to do data analysis on a salary data, the data path is: /Users/xxx/Downloads/Salary_Data.csv. Statistics on the average salary of different age groups, draw a line chart to visualize it.

Planning

```
[  
  {  
    "task_id": "1",  
    "dependent_task_ids": [],  
    "instruction": "Load the salary data from the CSV file.",  
    "task_type": "eda"  
  },  
  {  
    "task_id": "2",  
    "dependent_task_ids": ["1"],  
    "instruction": "Analyze the average salary by different age groups.",  
    "task_type": "eda"  
  },  
  {  
    "task_id": "3",  
    "dependent_task_ids": ["2"],  
    "instruction": "Draw a line chart to visualize the average salary by age groups.",  
    "task_type": "eda"  
}
```

Solving

Task1 Code

```
import pandas as pd  
import numpy as np  
  
# Load the salary data from the CSV file  
file_path = '/Users/xxx/Downloads/Salary_Data.csv'  
salary_data = pd.read_csv(file_path)  
  
# Display the first few rows of the dataset to understand its structure  
salary_data.head()
```

```
=====  
# Grouping the data by age and calculating the average salary for each age group  
age_salary_analysis = salary_data.groupby('Age')[['Salary']].mean().reset_index()  
  
# Importing necessary libraries for visualization  
import matplotlib.pyplot as plt  
  
# Plotting the average salary by age  
plt.figure(figsize=(10, 6))  
plt.plot(age_salary_analysis['Age'], age_salary_analysis['Salary'], marker='o')  
plt.title('Average Salary by Age Group')  
plt.xlabel('Age')  
plt.ylabel('Average Salary')  
plt.grid()  
plt.xticks(np.arange(min(age_salary_analysis['Age']), max(age_salary_analysis['Age']) + 1, 1))  
plt.show()
```

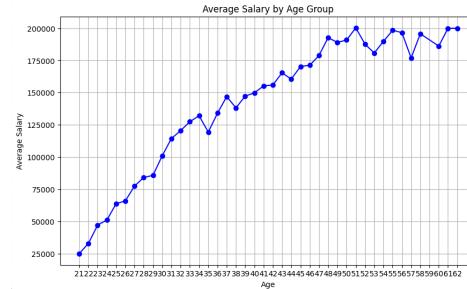
Results

=====Task1 Result=====

Age	Gender	Education Level	Job Title	Years of Experience	
0	32.0	Male	Bachelor's	Software Engineer	5.0
1	28.0	Female	Master's	Data Analyst	3.0
2	45.0	Male	PhD	Senior Manager	15.0
3	36.0	Female	Bachelor's	Sales Associate	7.0
4	52.0	Male	Master's	Director	20.0

Salary
0 90000.0
1 65000.0
2 150000.0
3 60000.0
4 200000.0

=====Task2 & 3 Result=====



Reflection

```
[  
  {  
    "task_id": "1",  
    "dependent_task_ids": [],  
    "instruction": "Load the salary data from the CSV file.",  
    "task_type": "eda",  
    "code": "...",  
    "result": "...",  
    "is_success": true,  
    "is_finished": true  
  },  
  {  
    "task_id": "2",  
    "dependent_task_ids": [  
      "1"  
    ],  
    "instruction": "Analyze the average salary by different age groups.",  
    "task_type": "eda",  
    "code": "...",  
    "result": "...",  
    "is_success": true,  
    "is_finished": true  
  },  
  {  
    "task_id": "3",  
    "dependent_task_ids": [  
      "2"  
    ],  
    "instruction": "Draw a line chart to visualize the average salary by age groups.",  
    "task_type": "eda",  
    "code": "...",  
    "result": "...",  
    "is_success": true,  
    "is_finished": true  
  }  
]
```

Current Task {}

Figure 8: Data Interpreter for data visualization.

task is considered finished.

Then, we used the Data Interpreter to train a classifier for breast cancer prediction based on the Breast Cancer Wisconsin (Diagnostic) Dataset³. We prompted the Data Interpreter to compute the classification accuracy using 5-fold cross-validation. Figure 9 illustrates the detailed prompt and problem-solving process.

Similarly, the Data Interpreter planned the whole task in 3 sub-tasks: (1) “Load and preprocess the breast cancer dataset”, (2) “Train a classifier using 5-fold cross-validation.”, and (3) “Evaluate the model and print the final accuracy.”. However, there was an error caused by the wrong column name when solving the first task. The Data Interpreter reflected the error and updated the task code. Eventually, it successfully finished all 3 sub-tasks and provided the final accuracy of 0.9649.

However, if the user’s prompt is relatively simple, such as tasks that could be completed in a single step, the end-to-end data agent may waste many tokens due to its multi-step decomposition process.

Furthermore, because the entire process cannot be intervened by the user, if any step produces an undesired outcome, the entire workflow must be repeated during the next session. This will result in the waste of time and tokens. Additionally, users often overlook some details in their prompts. For example, in this case, we asked the agent to train a model but forgot to instruct it to save the model. As a result, the model was not saved, and the entire process needs to be repeated in the next session. Moreover, without setting a fixed `random_state`, it is likely that we cannot reproduce the same model, leading to further inconsistencies.

To tackle this, the Data Interpreter offers another conversational mode for human interaction by specifying `--auto_run False`. This hybrid approach, which supports both

³<https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data>

(Q): I want to train a classifier to classify breast cancer using the data from /Users/xxx/Downloads/breast_cancer_wisconsin_diagnostic.csv. The target variable is 'diagnosis'. Use any model you think is appropriate. Perform 5-fold cross-validation during training and print the final accuracy.

Planning

```
[ {
    "task_id": "1",
    "dependent_task_ids": [],
    "instruction": "Load and preprocess the breast cancer dataset.",
    "task_type": "data preprocessing"
},
{
    "task_id": "2",
    "dependent_task_ids": ["1"],
    "instruction": "Train a classifier using 5-fold cross-validation.",
    "task_type": "model train"
},
{
    "task_id": "3",
    "dependent_task_ids": ["2"],
    "instruction": "Evaluate the model and print the final accuracy.",
    "task_type": "model evaluate"
}]
```

Solving Task1

```
=====Task1 Code=====
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = pd.read_csv('/Users/xxx/Downloads/breast_cancer_wisconsin_diagnostic.csv')

# Monitor data types per column
print(data.dtypes)

# Copy the DataFrame for processing
data_copy = data.copy()

# Check for missing values
print(data_copy.isnull().sum())

# Encode the target variable 'diagnosis' (M=1, B=0)
data_copy['diagnosis'] = data_copy['diagnosis'].map({'M': 1, 'B': 0})

# Identify features and target
X = data_copy.drop(columns=['diagnosis'])
y = data_copy['diagnosis']

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Display the shapes of the resulting datasets
(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

Results

ID	int64	ID	0
radius1	float64	radius1	0
texture1	float64	texture1	0
perimeter1	float64	perimeter1	0
area1	float64	area1	0
smoothness1	float64	smoothness1	0
compactness1	float64	compactness1	0
.....			

✗ **KeyError: 'diagnosis'**
Error Information

Error Reflection

It seems that there is a discrepancy in the column name for the target variable. The column is labeled as 'Diagnosis' (with a capital 'D') instead of 'diagnosis'. Let's correct this and proceed with the preprocessing steps again.

Here's the updated code for the current task:

```
=====Task1 Updated Code=====
# Encode the target variable 'Diagnosis' (M=1, B=0)
data_copy['Diagnosis'] = data_copy['Diagnosis'].map({'M': 1, 'B': 0})

# Identify features and target
X = data_copy.drop(columns=['Diagnosis'])
y = data_copy['Diagnosis']

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Display the shapes of the resulting datasets
(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

Results

```
=====Task1 Result=====
((455, 31), (114, 31), (455,), (114,))
```

Solving Task2

```
=====Task2 Code=====
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

# Initialize the classifier
classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Perform 5-fold cross-validation
cv_scores = cross_val_score(classifier, X_scaled, y, cv=5)

# Calculate and print the final accuracy
final_accuracy = np.mean(cv_scores)
print(f'Final Accuracy from 5-fold Cross-Validation: {final_accuracy:.4f}')
```

Results

```
=====Task2 Result=====
Final Accuracy from 5-fold Cross-Validation: 0.9578
```

Solving Task3

```
=====Task3 Code=====
# Evaluate the model using the test set
classifier.fit(X_train, y_train) # Fit the model on the training data
test_accuracy = classifier.score(X_test, y_test) # Evaluate on the test set
print(f'Final Accuracy on Test Set: {test_accuracy:.4f}')
```

Results

```
=====Task3 Result=====
Final Accuracy on Test Set: 0.9649
```

Figure 9: End-to-end case of machine learning by Data Interpreter.

end-to-end and conversational modes, is likely to become a prevailing design trend in the future.

4.3 Case study 3: Explore Expandability in Data Agents

In many situations, we encounter tasks that cannot be handled effectively because the LLM lacks the necessary knowledge. In such cases, if the data agent is designed to be extensible, manual tool expansion or knowledge integration can address this limitation. In this case study, we demonstrate how both the Data Interpreter and LAMBDA leverage integration mechanisms to incorporate additional packages or domain-specific knowledge.

Creating and Using Tools in Data Interpreter To integrate a tool in MetaGPT (the parent project of Data Interpreter), we need to create a function or class in the `metagpt/tools/libs` directory and annotate it with the `@register_tool` decorator. This registration makes the tool accessible to the Data Interpreter. Besides, functions and classes must also include Google-style docstrings to describe their purpose and usage. More details can be found at document⁴. In this case, we used the web-scraping tool `PlaywrightWrapper` to gather deadlines for AI conferences from the website⁵. In the prompt, we included the URL of the target site and specified that the scraped data should be saved in a `.txt` file. The tool analyzed the HTML structure of the page and extracted relevant information, such as conference names and submission deadlines. Figure 10 illustrates the detailed prompt, tool creation, and tool usage in solving the problem.

In the case, the Data Interpreter began with an initial plan. For each sub-task, it recommended relevant tools with a score indicating their suitability. The system then decided whether to use the suggested tool. For instance, it used `scrape_web_playwright`

⁴https://docs.deepwisdom.ai/main/en/guide/tutorials/create_and_use_tools.html

⁵<https://aideadlin.es>

<p>Tool Configuration</p> <pre> from metagpt.tools.tool_registry import register_tool from metagpt.tools.web_browser_engine_playwright import PlaywrightWrapper @register_tool(tags=["web scraping", "web"]) async def scrape_web_playwright(url): Asynchronously Scrape and save the HTML structure and inner text content of a web page using Playwright. Args: url (str): The main URL to fetch inner text from. Returns: dict: The inner text content and HTML structure of the web page, keys are 'inner_text', 'html' """ # Create a PlaywrightWrapper instance for the Chromium browser web = await PlaywrightWrapper().run(url) # Return the inner text content of the web page return {"inner_text": web.inner_text.strip(), "html": web.html.strip()} </pre> <p>Prompt</p> <pre> Fetch conference deadline from the website https://aideadlin.es/?sub=NLP, save it as a txt file. **Notice: Firstly parse the web page encoding and the text HTML structure; then save it in the txt file.** </pre> <p>Solving Planning</p> <pre> [{"task_id": "1", "dependent_task_ids": [], "instruction": "Fetch the conference deadlines from the NLP section of the website", "task_type": "web scraping"}, {"task_id": "2", "dependent_task_ids": ["1"], "instruction": "Parse the fetched data to extract the relevant deadlines", "task_type": "other"}, {"task_id": "3", "dependent_task_ids": ["2"], "instruction": "Save the extracted deadlines into a txt file", "task_type": "other"}] </pre> <p>Tools Recommend</p> <pre> 2024-11-15 02:48:39.841 INFO metagpt.tools.tool_recommend.recall_tools:195 - Recalled tools: [{"scrape_web_playwright": Scores: [-7.781837044732448], "json": [{"scrape_web_playwright": ... </pre>	<p>Tool Use</p> <pre> 2024-11-15 02:48:40.976 INFO metagpt.tools.tool_recommend.recommend_tools:101 - Recommended tools: [{"scrape_web_playwright"] # Import the necessary tool for web scraping from metagpt.tools.libs.web_scraping import scrape_web_playwright # Define the URL for the NLP section of the conference deadlines url = "https://aideadlin.es/?sub=NLP" # Asynchronously scrape the web page to get the inner text and HTML structure data = await scrape_web_playwright(url) # Extract the inner text content inner_text = data['inner_text'] # Save the extracted deadlines to a text file with open('conference_deadlines.txt', 'w') as file: file.write(inner_text) # Display a message indicating that the deadlines have been saved print("Conference deadlines have been saved to 'conference_deadlines.txt'") </pre> <p>Solving Task 2 and 3</p> <pre> Task2 & 3 Code import re # Assuming 'data' contains the fetched HTML data from the previous task # Extracting deadlines using regex or BeautifulSoup from bs4 import BeautifulSoup # Parse the HTML content soup = BeautifulSoup(data['html'], 'html.parser') # Find all deadlines (assuming they are in a specific tag, e.g., <i> or <div>) deadlines = [] for item in soup.find_all(['i', 'div']): if re.search(r'(d1 d2)-(d1 d2)', item.text): # Adjust the tag based on actual HTML structure if item.get('text'): if re.search(r'(d1 d2)-(d1 d2)', item['text']): # Looking for date patterns deadlines.append(item['text']) # Save the extracted deadlines to a text file with open('conference_deadlines.txt', 'a') as file: for deadline in deadlines: file.write(deadline + '\n') print("Extracted conference deadlines have been saved to 'conference_deadlines.txt'") </pre> <p>Result</p> <pre> conference_deadlines.txt Most recent first NAACL 2025 Deadline: Wed Oct 16 2024 19:59:59 GMT+0800 April 29 - May 4, 2024, Albuquerque, New Mexico, USA. Note: All submissions must be done through ARR. More info here. natural language proc ICLR 2025 Deadline: Wed Oct 02 2024 19:59:59 GMT+0800 Apr 24-28, 2025, Singapore Note: Mandatory abstract deadline on September 27, 2024. More info here. machine learning automated planning robotics computer vision natural language proc speech/sigproc COLING 2025 Deadline: Sat Sep 17 2024 19:59:59 GMT+0800 January 19-24, 2025, Abu Dhabi, UAE. Note: More info can be found here. natural language proc NeurIPS 2024 Deadline: Mon Jun 10 2024 19:59:59 GMT+0800 November 13-15, 2024, Montevideo, Uruguay machine learning natural language proc computer vision NeurIPS [Datasets and Benchmark Track] 2024 Deadline: Thu Nov 06 2024 19:59:59 GMT+0800 December 9 - December 15, 2024, Vancouver, Canada Note: Mandatory abstract deadline on May 29, 2024, and supplementary material deadline on June 12, 2024. More info here. data mining machine learning natural language proc speech/sigproc computer vision AAAI 2025 Deadline: Fri Aug 16 2024 19:59:59 GMT+0800 February 25 - March 04, 2025, Philadelphia, USA. Note: Mandatory abstract deadline on Aug 07, 2024, and supplementary material deadline on Aug 19, 2024. More info here. natural language representation machine learning robotics natural language proc speech/sigproc computer vision computer graphics human-computer interaction automated planning KIL 2024 Deadline: Sun May 26 2024 19:59:59 GMT+0800 August 25, 2024, Barcelona, Spain natural language proc machine learning data mining EMNLP 2024 Deadline: Sun Jun 16 2024 19:59:00 GMT+0800 July 12-19, 2024, Miami, Florida, USA Note: More info here. natural language proc ACL 2024 Deadline: Fri Feb 16 2024 19:59:00 GMT+0800 August 17, 2024, Bangkok, Thailand Note: All submissions should be done through ARR. More info here. natural language proc COLM 2024 Deadline: Sun Mar 30 2024 19:59:59 GMT+0800 October 12-19, 2024, Philadelphia, USA Note: Mandatory abstract deadline on March 22, 2024. More info here. natural language proc </pre>
--	---

Figure 10: Creating and using the customized tool in the Data Interpreter.

for a web-scraping task. This iterative recommendation and tool selection process continued until all sub-tasks were completed, addressing limitations in LLMs’ built-in abilities and knowledge.

Next, we demonstrate the knowledge integration mechanism in LAMBDA through Fixed Point Non-Negative Neural Networks (FPNNNs). First, we copied the code for FPNN into LAMBDA’s template, which automatically integrates the code into the knowledge base. Since the code was lengthy, we defined the mode as **Core**. We then delineated the **Core** function, which directly accepts parameters, and the Runnable function, which was defined and executed separately. Figure 7 presents the configuration, prompt, and problem-solving process.

LAMBDA first retrieved the relevant code from the knowledge base, and then its **Core** function was presented in the context. By modifying the core code, LAMBDA generated the correct code and completed the task successfully.

Knowledge Configuration

```

name: 'Fixed points of nonnegative neural networks'
description: 'This is fixed_points_of_nonnegative_neural_networks which used fixed point theory to analyze nonnegative neural networks, which we define as neural networks that map nonnegative vectors to nonnegative vectors. Variables: networks: nn_sigmoid, learning rate: 5e-3, epochs: 30, wd: 0, b: 64'
mode = 'core'
core_function: 'core'
runnable_function = 'runnable'
test_case = 'case_nn_networks'

case = """
args = argparse.ArgumentParser()
args.net = 'nn_sigmoid'
args.lr = 5e-3
args.epochs = 30
args.wd = 0
args.b = 64
train_nn_network(args)
"""

code = """
import numpy as np
import scipy.io as sio
import scipy
import sys
import time
import argparse
import torch
import math
from torch import nn
from torch.nn.utils.parametrize import spectral_norm
from pathlib import Path
from torch import optim
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
from tqdm import tqdm

def initialize_weights(tensor):
    return tensor.uniform_() * math.sqrt(0.25 / (tensor.shape[0] + tensor.shape[1]))

class _RRAutoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_1 = nn.Linear(784, 200)
        self.linear_2 = nn.Linear(200, 784)
        self.encoder = self.linear_1
        self.decoder = self.linear_2

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)

    return x
.....

```

Result

(Q): I want to train a fixed points non-negative neural network. We define as neural networks that map nonnegative vectors to nonnegative vectors. Use nn_sigmoid as the networks, set learning rate to 5e-3, run 5 epochs, set wd to 0 and b to 64. Print train and evaluation loss for each epoch. Use the MNIST dataset for training and testing.

(A)



Retrieval:
The retriever found the following pieces of code cloud address the problem. All functions and classes have been defined and executed in the back-end.

Core code (All functions and classes have been defined, directly use them in your code):

```

'''core_function
args = argparse.ArgumentParser()
args.net = 'nn_sigmoid'
args.lr = 5e-3
args.epochs = 30
args.wd = 0
args.b = 64
train_nn_network(args)
'''
```

Your code:

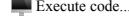
```

'''python
import argparse
```

```

# Set up the arguments
args = argparse.ArgumentParser()
args.net = 'nn_sigmoid'
args.lr = 5e-3
args.epochs = 5
args.wd = 0
args.b = 64
```

Train the neural network
train_nn_network(args)
'''



Execute code...

Execution result:
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz

Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw

Epoch: 1 | Train Loss: 0.2316 | Test Loss: 0.2311

Epoch: 2 | Train Loss: 0.2310 | Test Loss: 0.2309

Epoch: 3 | Train Loss: 0.2309 | Test Loss: 0.2309

Epoch: 4 | Train Loss: 0.2309 | Test Loss: 0.2309

Epoch: 5 | Train Loss: 0.2309 | Test Loss: 0.2309

```

'''markdown
| Epoch | Train Loss | Test Loss |
|-----|-----|-----|
| 1 | 0.2316 | 0.2311 |
| 2 | 0.2310 | 0.2309 |
| 3 | 0.2309 | 0.2309 |
| 4 | 0.2309 | 0.2309 |
| 5 | 0.2309 | 0.2309 |
'''
```

The training and test losses show a decreasing trend, indicating that the model is learning and performing well on the MNIST dataset. As a next step, you may want to evaluate the model's performance using classification metrics, such as accuracy or F1-score, on the test set. Additionally, you could consider experimenting with different architectures or hyperparameters to further improve the performance.

Figure 11: Integrating knowledge of FPNNs in LAMBDA.

Effective integration of specialized tools and knowledge into data agents can be achieved with the help of advanced tools. Leveraging large language models (LLMs) reduces the need for domain experts to cover every aspect of development, such as when creating a Python package. This approach simplifies the traditional development process to some extent.

5 Challenges and Future Directions

Despite the recent advancements in data analysis introduced by various data science agents, many challenges remain. Here, we highlight some of these challenges and suggest future directions for the field.

5.1 Challenges in the Capabilities of LLMs

The LLMs serve as the “brain” of a data agent, interpreting user intent and generating structured plans to execute data analysis tasks. For an ideal data agent, advanced knowledge in statistics, data science, and coding is essential to effectively support users throughout the analytical process.

Advanced Models and Planning Techniques Current state-of-the-art models like GPT-4 show strong performance on undergraduate-level mathematics and statistics problems, yet struggle with more advanced, graduate-level tasks (Frieder et al., 2023). Additionally, the success rate of fully automating complete data workflows with current agents remains low (Cao et al., 2024). This suggests that enhancements in domain-specific knowledge, particularly in statistics and data analysis, are still needed, along with specialized planning and reasoning skills for data agents. Furthermore, to accommodate user preferences, data agents must be capable of learning from past interactions, requiring advancements in memory storage and adaptive systems (Inala et al., 2024).

Multi-Modality and Reasoning Current data agents also struggle with handling multi-modal data—such as charts, tables, and code—which are crucial for data analysis workflows (Inala et al., 2024). Future research might focus on utilizing multi-modality models, like VLMs, to enhance reasoning capabilities across diverse data formats, allowing for a deeper understanding of relationships within and between various data representations (Inala et al., 2024). For instance, if a user finds a well-designed chart online with specific aesthetics and wishes to recreate it, describing these details verbally may be challenging. A multi-modal data agent, however, could allow the user to upload the image and replicate the chart’s design by generating the necessary code based on the visual input. This requires more advanced VLMs and planning techniques to achieve these goals. For instance, a color extractor could be utilized during the process to extract hex color codes from specific visual elements.

5.2 Challenges in Statistical Analysis

Intelligent Statistical Analysis Software Current statistical software, like SPSS, and programming languages such as R, are highly developed, but future data agents have the potential to evolve into a new paradigm of intelligent statistical analysis software. This vision presents several challenges. For instance, R possesses a vast community and extensive package ecosystem, enabling users to easily perform a wide range of tasks by package installation. For data agents to succeed as intelligent analysis tools, they must facilitate seamless package development and installation, foster a large community where domain experts can contribute, and stay updated with knowledge from various programming communities. This collaborative environment could accelerate the evolution and experimentation within intelligent statistical analysis software. Additionally, many statisticians and scientists may

lack comprehensive knowledge of existing models and methods, so data agents that offer guidance and suggest relevant approaches could significantly enhance research efficiency and drive scientific innovation.

Incorporating Other Large Models into Statistical Analysis Current statistical analysis may increasingly rely on large models’ representations for research. For example, in predicting the tertiary structure of proteins, LLMs can utilize representations of primary and secondary structures. Traditional statistical software like Matlab and R currently lacks robust applications in these areas. If data agents can effectively leverage domain-specific knowledge models, they could significantly advance statistical and data science research, enabling more complex analyses and fostering deeper insights across scientific disciplines.

Other challenges related to data infrastructure and the evaluation of data agents have been addressed in prior works (Inala et al., 2024) and will not be discussed further in this survey. Additionally, for web-based applications, issues such as handling high concurrent requests present notable challenges. Effectively managing resource allocation and scheduling across multiple sandbox environments is crucial to ensuring scalability and efficiency, making these topics important areas for future research and development.

6 Conclusion

This survey has explored the recent progress of LLM-based data science agents. These agents have shown great potential in making data analysis more accessible to a wider range of users, even those with limited technical skills. By leveraging the capabilities of LLMs, they are able to handle various data analysis tasks, from data visualization to machine learning, through natural language interaction.

However, as discussed, they also face several challenges. In terms of model capabilities,

improvements are needed in domain-specific knowledge and multi-modal handling. For intelligent statistical analysis software, seamless package management and community building are crucial. Additionally, effectively integrating other large models into statistical analysis and addressing data infrastructure and evaluation issues remain important areas for future development.

Overall, while LLM-based data science agents have made significant strides, continuous research and innovation are required to overcome the existing challenges and fully realize their potential in revolutionizing the field of data analysis.

References

- Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawska, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczek, P., et al. (2024). Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.
- Cao, L. (2017). Data science: Challenges and directions. *Communications of the ACM*, 60(8):59–68.
- Cao, R., Lei, F., Wu, H., Chen, J., Fu, Y., Gao, H., Xiong, X., Zhang, H., Mao, Y., Hu, W., Xie, T., Xu, H., Zhang, D., Wang, S., Sun, R., Yin, P., Xiong, C., Ni, A., Liu, Q., Zhong, V., Chen, L., Yu, K., and Yu, T. (2024). Spider2-v: How far are multimodal agents from automating data science and engineering workflows?
- Chandel, S., Clement, C. B., Serrato, G., and Sundaresan, N. (2022). Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901*.
- chapyter (2023). Chapyter. <https://github.com/chapyter/chapyter>.
- Chen, H., Chiang, R. H., and Storey, V. C. (2012). Business intelligence and analytics: From big data to big impact. *MIS quarterly*, 36(4):1165–1188.
- Chen, Z., Cao, L., Madden, S., Kraska, T., Shang, Z., Fan, J., Tang, N., Gu, Z., Liu, C., and Cafarella, M. (2024). Seed: Domain-specific data curation with large language models. arxiv 2023. *arXiv preprint arXiv:2310.00749*.
- Cheng, L., Li, X., and Bing, L. (2023). Is gpt-4 a good data analyst? *arXiv preprint arXiv:2305.15038*.

- Chi, Y., Lin, Y., Hong, S., Pan, D., Fei, Y., Mei, G., Liu, B., Pang, T., Kwok, J., Zhang, C., et al. (2024). Sela: Tree-search enhanced llm agents for automated machine learning. *arXiv preprint arXiv:2410.17238*.
- Dash, T., Chitlangia, S., Ahuja, A., and Srinivasan, A. (2022). A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Scientific Reports*, 12(1):1040.
- Dong, H. and Wang, Z. (2024). Large language models for tabular data: Progresses and future directions. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 2997–3000, New York, NY, USA. Association for Computing Machinery.
- for Statistical Computing, R. F. (1995). *R: A Language and Environment for Statistical Computing*.
- Foundation, P. S. (1991). *Python Programming Language*.
- Frieder, S., Pinchetti, L., Chevalier, A., Griffiths, R.-R., Salvatori, T., Lukasiewicz, T., Petersen, P. C., and Berner, J. (2023). Mathematical capabilities of chatgpt.
- GLM, T. (2024). Chatglm: A family of large language models from glm-130b to glm-4 all tools.
- Grosnit, A., Maraval, A., Doran, J., Paolo, G., Thomas, A., Beevi, R. S. H. N., Gonzalez, J., Khandelwal, K., Iacobacci, I., Benechehab, A., et al. (2024). Large language models orchestrating structured reasoning achieve kaggle grandmaster level. *arXiv preprint arXiv:2411.03562*.
- Gu, K., Shang, R., Althoff, T., Wang, C., and Drucker, S. M. (2024). How do analysts understand and verify ai-assisted data analyses? In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–22.
- Guo, S., Deng, C., Wen, Y., Chen, H., Chang, Y., and Wang, J. (2024). Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*.
- Hassan, M. M., Knipper, A., and Santu, S. K. K. (2023). Chatgpt as your personal data scientist. *arXiv preprint arXiv:2305.13657*.
- Hong, S., Lin, Y., Liu, B., Wu, B., Li, D., Chen, J., Zhang, J., Wang, J., Zhang, L., Zhuge, M., et al. (2024). Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*.
- Huang, X., Liu, W., Chen, X., Wang, X., Wang, H., Lian, D., Wang, Y., Tang, R., and Chen, E. (2024). Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*.
- IBM (1968). *SPSS Statistics*.

- Inala, J. P., Wang, C., Drucker, S., Ramos, G., Dibia, V., Riche, N., Brown, D., Marshall, D., and Gao, J. (2024). Data analysis in the era of generative ai. *arXiv preprint arXiv:2409.18475*.
- Inc., S. I. (1976). *SAS Software*.
- Institute, M. G. (2011). *Big data: The next frontier for innovation, competition, and productivity*. McKinsey & Company.
- Jiang, Z. et al. (2024). AIDE: the Machine Learning CodeGen Agent. <https://github.com/WecoAI/aideml>. Accessed: 2024-08-29.
- Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.
- jupyterlab (2023). Jupyter-ai. <https://github.com/jupyterlab/jupyter-ai>.
- Kitchin, R. (2014). *The data revolution: Big data, open data, data infrastructures and their consequences*. Sage.
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.
- Li, Z., Zang, Q., Ma, D., Guo, J., Zheng, T., Niu, X., Yue, X., Wang, Y., Yang, J., Liu, J., et al. (2024). Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*.
- Liu, S.-C., Wang, S., Chang, T., Lin, W., Hsiung, C.-W., Hsieh, Y.-C., Cheng, Y.-P., Luo, S.-H., and Zhang, J. (2023). Jarvix: A llm no code platform for tabular data analysis and optimization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 622–630.
- Luo, D., Feng, C., Nong, Y., and Shen, Y. (2024). Autom3l: An automated multimodal machine learning framework with large language models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 8586–8594.
- Ma, P., Ding, R., Wang, S., Han, S., and Zhang, D. (2023). Insightpilot: An llm-empowered automated data exploration system. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 346–352.
- MathWorks (1984). *MATLAB*.
- Microsoft (1985). *Microsoft Excel*.
- Microsoft (2013). *Power BI*.
- Nejjar, M., Zacharias, L., Stiehle, F., and Weber, I. (2023). Llms for science: Usage for code generation and data analysis. *Journal of Software: Evolution and Process*, page e2723.

- OpenAI (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Provost, F. and Fawcett, T. (2013). Data science and its relationship to big data and data-driven decision making. *Big data*, 1(1):51–59.
- Qiao, B., Li, L., Zhang, X., He, S., Kang, Y., Zhang, C., Yang, F., Dong, H., Zhang, J., Wang, L., et al. (2023). Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.
- Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. (2024). Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.
- Steffensen, J. L., Dufault-Thompson, K., and Zhang, Y. (2016). Psamm: A portable system for the analysis of metabolic models. *PLOS Computational Biology*, 12(2):1–29.
- Sun, M., Han, R., Jiang, B., Qi, H., Sun, D., Yuan, Y., and Huang, J. (2024). Lambda: A large model based data agent. *arXiv preprint arXiv:2407.17535*.
- Trirat, P., Jeong, W., and Hwang, S. J. (2024). Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*.
- Tu, X., Zou, J., Su, W. J., and Zhang, L. (2023). What should data science education do with large language models?
- Waller, M. A. and Fawcett, S. E. (2016). Data science, predictive analytics, and big data: A revolution that will transform supply chain design and management. *Journal of Business Logistics*, 37(1):55–62.
- Wang, C., Lee, B., Drucker, S., Marshall, D., and Gao, J. (2024a). Data formulator 2: Iteratively creating rich visualizations with ai. *arXiv preprint arXiv:2408.16119*.
- Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. (2024b). Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Witten, I. H., Frank, E., and Hall, M. A. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
- Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al. (2023). The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.
- Xie, L., Zheng, C., Xia, H., Qu, H., and Zhu-Tian, C. (2024). Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization. *arXiv preprint arXiv:2408.01703*.

- Xie, T., Zhou, F., Cheng, Z., Shi, P., Weng, L., Liu, Y., Hua, T. J., Zhao, J., Liu, Q., Liu, C., et al. (2023). Openagents: An open platform for language agents in the wild. *arXiv preprint arXiv:2310.10634*.
- Yang, Z., Zhou, Z., Wang, S., Cong, X., Han, X., Yan, Y., Liu, Z., Tan, Z., Liu, P., Yu, D., et al. (2024). Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *arXiv preprint arXiv:2402.11453*.
- Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. (2024). Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.
- Zhang, C., Li, L., He, S., Zhang, X., Qiao, B., Qin, S., Ma, M., Kang, Y., Lin, Q., Rajmohan, S., et al. (2024). Ufo: A ui-focused agent for windows os interaction. *arXiv preprint arXiv:2402.07939*.
- Zhang, L., Zhang, Y., Ren, K., Li, D., and Yang, Y. (2023a). Mlcopilot: Unleashing the power of large language models in solving machine learning tasks. *arXiv preprint arXiv:2304.14979*.
- Zhang, W., Shen, Y., Lu, W., and Zhuang, Y. (2023b). Data-copilot: Bridging billions of data and humans with autonomous workflow. *arXiv preprint arXiv:2306.07209*.
- Zhou, X., Zhao, X., and Li, G. (2024). Llm-enhanced data management. *arXiv preprint arXiv:2402.02643*.