CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

TableAnalyst: an LLM-agent for tabular data analysis.

Implementation and evaluation on tasks of varying complexity.

Master's thesis in Applied Data Science

Andris Freimanis

Patrick Andersson Rhodin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

TableAnalyst: an LLM-agent for tabular data analysis.

Implementation and evaluation on tasks of varying complexity.

Andris Freimanis, Patrick Andersson Rhodin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

TableAnalyst: an LLM-agent for tabular data analysis.
Implementation and evaluation on tasks of varying complexity.
Andris Freimanis, Patrick Andersson Rhodin

© Andris Freimanis, Patrick Andersson Rhodin 2024.

Supervisor: Yinan Yu, Department of Computer Science and Engineering
Advisor: Dhasarathy Parthasarathy, Volvo Group, Sweden
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

TableAnalyst: an LLM-agent for tabular data analysis.
Implementation and evaluation on tasks of varying complexity.
Andris Freimanis, Patrick Andersson Rhodin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The automotive industry relies on manual analysis of tabular data, particularly for software test result evaluation. This time-consuming process diverts engineers from their core expertise. This study explores the application of a large language model (LLM) agent to address this challenge. We present a prototype LLM agent specifically designed for analyzing software test data in the automotive industry. Our agent employs an iterative feedback approach, utilizing two key modules: a planner and a code interpreter (CI). The planner decomposes user queries into step-by-step plans for data manipulation, while the CI translates these steps into executable code. To evaluate the agent's output quality, we introduce a novel dataframe similarity metric validated by a domain expert. This metric demonstrates promise as a valuable tool for evaluating table analysis agents. Furthermore, we explore task complexity metrics through correlation analysis. The results suggest that LLMs can identify complexity within the analysis query. However, the results also suggest that LLMs may struggle to translate this query into actionable plan steps. Overall, this work demonstrates the potential of LLM agents for automating software test data analysis tasks in the automotive industry. The development of complexity metrics and a novel evaluation metric contributes to further research and improvement of such agents.

Keywords: LLM, LLM-agent, tabular data, analysis, evaluation, task complexity.

Acknowledgements

First and foremost, a sincere thank you to our supervisor Yinan Yu, who time and time again cleared up confusions and kept us on the path towards project completion. She was the spark that ignited our fire, the guiding light in our darkness, the shining star in the night sky. A sincere thank you also to Volvo Group, Sweden, for enabling this project, and to our company supervisor Dhasarathy Parthasarathy for his inspiration, insights and guidance. Taking part in day-to-day activities of the team has provided us with valuable experience.

A big thank you to Azdine Bellabes, who is the verification engineer domain expert that has been the source of all knowledge and insights regarding verification work and the practical problem this thesis tackles.

Patrick would also like to thank his wife and mother and father, without them his successful participation in this project would not have been possible.

Andris would like to thank himself not only for the dedication and perseverance poured into this thesis, but also for the long and arduous journey that led him here. The broad nature of the topic presented a constant challenge to stay focused, but he pushed through the obstacles, adapting his approach as needed. I am proud of the growth he has experienced throughout this demanding journey. Inspiring, it is a testament to the resilience of the human spirit.

Andris Freimanis & Patrick Andersson Rhodin, Gothenburg, 2024-06-20

Contents

List of Figures

List of Tables

1

Introduction

This thesis begins with a story about Thomas the Verification Engineer, a fictional employee in a large automotive corporation. Thomas' job is to verify embedded software on trucks as they pass through phases of development and before they are allowed to be driven on public roads. In large automotive corporations, research and development is always ongoing. Prototypes of new vehicle features are developed and tested frequently and the test-result data is gathered. Automated static analyses are performed and reported to stakeholders via a web interface. However, stakeholders often have questions that they do not easily find the answers to in this interface, or have a need for some ad-hoc analysis on the test data that go beyond the reports. Some stakeholders reach out to Thomas with their questions and Thomas sees the value in answering them in a timely manner, since the answers very well might give important insights. However, the answers are not always readily available to him in his role as an engineer, but require him to put down his verification and integration tools and switch to a data analyst role that instead wields Microsoft PowerBI [1].

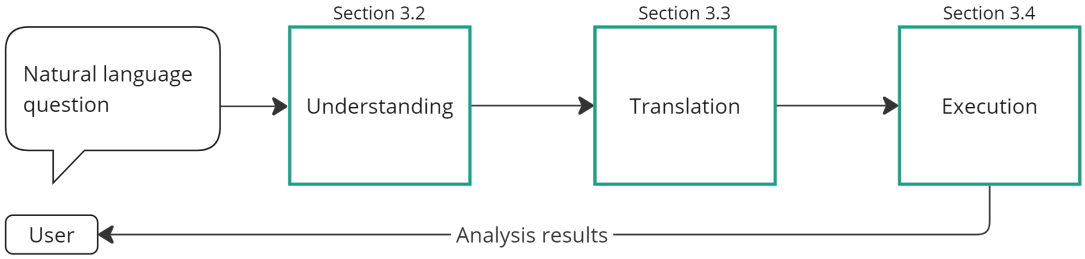
This switch of roles takes some time and temporarily removes him from his other day-to-day tasks, tasks that in comparison much more depend on his expertise. In fact, integration and verification work is continuous and Thomas handles about one full suite of tests every week. A test suite is likely to contain either about 90-100 or about 170-190 individual test cases[2] that need to be integrated and verified. Integration involves setting up the right electronic control units and other hardware and equipment, as well as downloading the correct software to each component. Verification means running the integrated system and checking and reporting the result of each test case. Sometimes the tests are performed on laboratory rigs and other times the rigs are installed in actual trucks. The demand for Thomas as an engineer does not pause while Thomas is working as a data analyst, and it is vital that tests are run in a timely manner to avoid delays in the software release cycle.

As such, Thomas the Verification Engineer collects the questions that come in from stakeholders and schedules time to work as Thomas the Data Analyst when the overhead of switching roles is reasonable, or when the demand for Thomas as an engineer is lower than usual. Obtaining the additional analyses can be time-consuming and does not contribute anything to filling the data, which is Thomas' main job as an engineer. There is no one that Thomas can delegate the analysis task to, and

[1] https://www.microsoft.com/en-us/power-platform/products/power-bi

[2] Estimated from testing data covering a period over 538 days during 2022-2023.

Figure 1.1: An overview of the automation problem addressed in this thesis. A natural language question needs to be understood and translated into actions, and those actions need to be executed. The sections referred to above the boxes cover the problem in more detail.



organizational changes are currently not an option. The workload does not warrant a dedicated analyst, and teaching the stakeholders to find the answers themselves has been deemed too difficult[3].

This thesis aims to provide a prototype tool for Thomas that reduces the time he spends as an analyst by orders of magnitude, enabling him to apply his expertise where it generates the most value to the company (i.e., filling the data and progressing the development cycle). This tool automates some of the data analysis tasks, from a natural language question about the data to an answer in the form of a human-readable table.

1.1 Automating ad-hoc tabular data analysis

Automating ad-hoc tabular data analysis requires a system that correctly interprets a natural language question, performs the correct analysis operations to arrive at data that answers the question, and report that result back to the user. This basic flow is illustrated in Figure 1.1. The first challenge is a natural language understanding (NLU)-task (A. Wang et al., 2018), addressed within the long-standing research field of natural language processing (NLP) (Joseph et al., 2016; Warren & Pereira, 1982). The second challenge is to translate this understanding of the question to *actions* that can act on the data to perform analysis. This entails the use of some tool(s) that can access and manipulate the data from a database or a file (in the case of this thesis: an Excel-workbook) such that the result can be returned as an answer to the question. A third challenge is to monitor the analysis process and intermediate results, since some analysis actions will depend on the result from earlier actions.

These challenges mirror the challenges that have faced the long-standing research

[3]Educational efforts have actually been tested but were abandoned. Stakeholders are very varied in organization and role which complicates communication and administration, and any educational efforts would need to be continuous, for example to educate stakeholders new to the job.

field of *intelligent agents* (Juliani et al., 2018; Wooldridge & Jennings, 1995). An intelligent agent in this regard refers to an artificial (i.e., non-organic) system that autonomously performs actions in an environment, in an intelligent manner. For an intelligent agent to autonomously perform tasks it needs to both perceive and understand its environment, and it needs to select and execute actions depending on what it learns from its percepts. For this the agent also needs some knowledge beyond what is embedded in the asked question. Classic agent theories have relied on symbolic methods[4] and reinforcement learning (RL) to address these challenges, making agents non-generalizable and difficult to implement (Russell & Norvig, 2021).

The field was recently invigorated (Xi et al., 2023) with the arrival and maturation of a new tool that seems to address all of these challenges, namely large language models (LLMs) (e.g., Brown et al., 2020; Chowdhery et al., 2023; Wei, Tay, et al., 2022; W. X. Zhao et al., 2023). LLMs are large, pre-trained, deep artificial neural networks based on the Transformer architecture (Vaswani et al., 2017), an architecture that has consistently set new state-of-the-art benchmarks on tasks across the field of NLP. LLMs also have other capabilities that address classical agent challenges regarding knowledge, planning, and action execution.

The pre-training is performed on vast amounts of text data from the internet, encoding not only language priors but also knowledge embedded in the language, providing a possible source of parametric knowledge for agents (Z. Zhao et al., 2024). Some LLMs also possess what seems like emergent reasoning capabilities that enable them to solve complex problems (Wei, Tay, et al., 2022), especially when guided via *prompt engineering* (P. Liu et al., 2021). In addition, most pre-trained LLMs are able to generate source code quite proficiently (Austin et al., 2021; Xu et al., 2022), providing a powerful tool for executing data analysis actions.

1.1.1 Large language model-based intelligent agents

Given this seemingly perfect match of LLM-capabilities and agent challenges, the interest in agents has exploded (Figure 1.2 shows search interest data from Google Trends.) and commercial and industrial investments are at an all time high. The now artificial intelligence (AI)-centered company Nvidia [5] has a research lab called GEAR [6] that focuses on LLMs for embodied agents, and Google has released their platform Vertex AI Agent Builder to offer a "no code"-solution to creating "generative AI experiences." (Gokturk, 2024) The market cap of NVIDIA stock has risen to about $2.9 trillion as of May 2024, from around $300 billion in October 2022, making them the third most valuable company in the world[7].

In the scientific literature there are both general and domain-specific agents, such as `TaskWeaver` (Qiao et al., 2023) and `WebGPT` (Nakano et al., 2021) respectively. `WebGPT` is an early agent that can answer questions by surfing the web that acquired

[4]Symbolic methods refer to methods that rely on human-crafted rules and representations, in contrast to statistical and connectionist methods that learn from data.

[5]https://www.nvidia.com/ Earlier known as a computer graphics company.

[6]https://research.nvidia.com/labs/gear/

[7]https://companiesmarketcap.com/nvidia/marketcap/

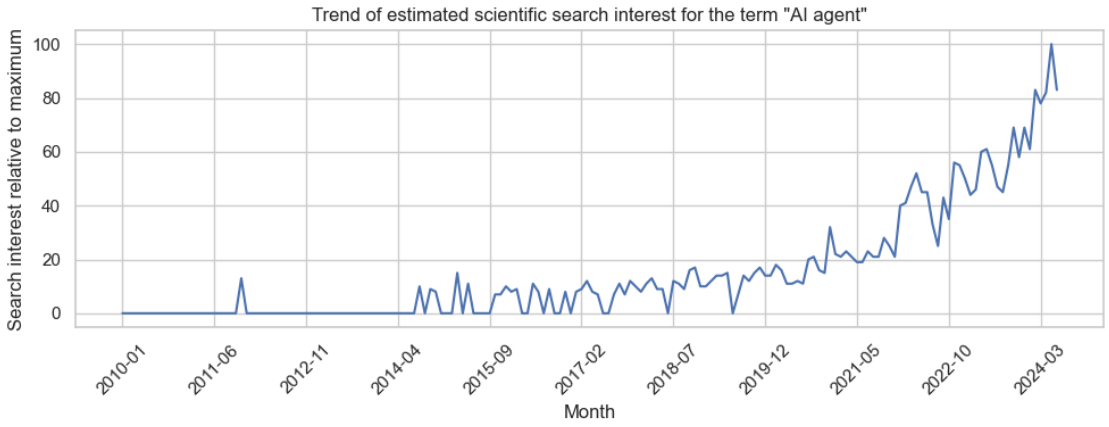


Figure 1.2: Search interest data from Google Trends for the term "ai agent" in category "science". The Transformer architecture was introduced in 2017, GPT-3 in May 2020, and ChatGPT was released November 2022.

its skills through RL, while `TaskWeaver` is an agent that ideally (after configuration for the new task) can do anything that can be done with Python code in a Jupyter environment. No fine-tuning required.

Since the capability acquisition paradigm has shifted from fine-tuning pre-trained models to a *pre-train, prompt, and predict* (P. Liu et al., 2021) paradigm, LLM-technology has become more accessible and research on agent-related topics has accelerated. Cheng et al. (2024) lists 34 different LLM-based single-agent systems and 11 multi-agent systems, developed for varied fields such as sociology, law, chemistry, mathematics, coding, and of course "universal" agents.

The state-of-the-art agent regarding data analysis is `Data Interpreter` (Hong et al., 2024), an agent that uses dynamic planning in a hierarchical graph structure, dynamic tooling, and experience gathering, to successfully solve a wide range of data science tasks. See Figure 3.3 to see a comparison of results between `Data Interpreter` and other similar agents.

`Data Interpreter` is a complex system that consists of many parts, and in addition to its own complexity, it is developed as a part of the multi-agent framework `MetaGPT` (Hong et al., 2023). In `MetaGPT`, `Data Interpreter` is implemented as a *role* among others, roles such as architect, engineer, product manager, searcher, and teacher. More on `Data Interpreter` and other LLM-based agents in Section 3.5. When implementing an agent in a real-world industry setting, the large code base and complexity of the framework is an important factor to consider.

1.2 Contributions of this thesis

To the best of our knowledge, there is no consensus or established literature on the implementation of a tabular data analyst LLM agent in a real-world industry setting. In contrast to research and public open-source projects, any implementation

in industry will need to consider factors other than achieving maximum success rate, factors such as available resources and existing in-house technology and code-bases. Agents from research literature focus mostly on maximizing performance (i.e., scoring high on general benchmarks, covered in Section 3.7) or making an agent as general as possible, bringing complexity that is not necessary for the specific task at hand. Being general often means that you lose specificity, and unnecessary complexity can reduce maintainability of the code base.

In the context of this thesis, an agent that solves *most* tasks given to it and is easy to further develop and maintain within existing in-house frameworks is preferred, to an agent that solves more tasks but also brings increased and unnecessary code complexity. As such, this thesis probes for the components and processes in an agent that will enable it to perform ad-hoc tabular data analysis, rather than probing for maximum success rate. As such we state our first research questions to guide the development:

> **RQ 1:** What is the most suitable LLM agent architecture for enabling effective ad-hoc tabular data analysis, considering task complexity and minimal human intervention?

To answer this we analyze processes from the agent literature with the tabular data analysis domain in mind, looking for the least common denominators between agents that claim success with such tasks. To consider task complexity means that the agent does not need components that are necessary for solving very complex tasks, such as developing a machine learning model. The agent only needs components that enable it to perform common tabular data analysis operations such as filtering and aggregating. To be able to numerically compare tasks of different complexity we apply a task complexity metric based on the components and relationships between steps in a solution.

We then implement a prototype agent based on the identified components and architecture. Our design choices, constraints, and implementation details are presented and discussed in Chapter 4.

To evaluate the agent we first look to the literature and review whether any existing benchmarks are applicable (see Section 3.7). Here we find that there are no benchmarks that would give a fair evaluation of our agent, either they are too general or they test capabilities that are irrelevant to our specific automation problem. As such, we develop our own evaluation dataset from use-cases given to us by a domain expert (see Section 4.1 for details).

In parallel we develop a metric to see how similar the agent's results are to the expected results defined by us in our evaluation dataset. Developing the dataset and designing experiments is done with our second and third research question in mind:

> **RQ 2:** How can we evaluate the performance of a domain-specific application of an LLM agent designed for analyzing software test data, considering the limitations of generic evaluation benchmarks?

RQ 3: How can analysis task complexity be measured from a natural language query, and how does it influence the performance of a domain-specific LLM agent for tabular data analysis?

2

Background

This chapter begins with a short discussion on an ethical aspect of LLM-technology, namely on the language the research community has adopted when describing what their inventions are doing. Words like "reasoning", "remembering", or "thinking" are not words that well represent the inner workings of next-token-predictors, and it might affect how non-experts and experts alike interpret the technology.

The chapter will then present a description of the actual tasks that the analyst agent is expected to solve, and a description of the dataset that analysis is to be performed on, followed by the practical constraints and choices that relate to this thesis and the collaboration with Volvo Group.

2.1 Intentional systems and anthropomorphism

Going forward there are going to be statements about agents and LLMs remembering, believing, understanding and reasoning, which are words we usually reserve to describe mental states of humans and other animals. This is not unusual and completely natural (Heider & Simmel, 1944), and would fall under what cognitive scientist Daniel Dennett calls *the intentional stance* (Dennett, 1989). Taking an intentional stance towards a machine is usually completely harmless and is also very useful to be able to understand and talk about it. As long as the subject of the intentional stance behaves in accordance with the theory behind the behavior, and we do not have a simpler way of describing the situation, then taking the intentional stance can help us understand, predict and even repair or improve the subject of the stance (Wooldridge & Jennings, 1995). However, taking the intentional stance with much simpler mechanisms becomes absurd:

> It is perfectly coherent to treat a light switch as a (very cooperative) agent with the capability of transmitting current at will, who invariably transmits current when it believes that we want it transmitted and not otherwise; flicking the switch is simply our way of communicating our desires. (Shoham, 1993)

Since we have a simpler, mechanistic, explanation of a light switch it does not make sense to involve wants and wills. However, "my computer does not want to talk to the network" or "my phone thinks we are in the water" makes perfect sense for most of us since we might not know or care about the underlying causes. Cognitively, the

same neural networks in our brains activate when assigning intentions to humans and non-human entities, with the only difference being activation in the prefrontal cortex to bypass the default pattern of assigning intentions to humans but not machines (Abu-Akel et al., 2020). The described situations so far are harmless since no one *actually* believes that we can fix the computer with a motivational speech, nor that the phone is *thinking* like a human would. One reason could be that these machines are not human-like enough to strongly activate the neural circuits involved in the recognition of actions performed by non-humans (Buccino et al., 2004)[1]

It is however wise to consider how we speak about machines that are human-like enough to strongly trigger such neural circuits, where our cognition does not always automatically discard the statements as just a manner of expression. In language generation LLMs already surpass human performance in some cases, such as in one case of writing argumentative essays for secondary education (Herbold et al., 2023). When it is easy to ascribe human-like characteristics to a machine, it can be irresponsible and even dangerous to misrepresent their inner workings (Ruane et al., 2019; Shanahan, 2024; Weidinger et al., 2021).

In this work we will present the inner workings of both LLMs and agent systems, in some detail, and try to be as specific as possible when referring to any human-like capacities. This way it will be clear what we are referring to when using anthropomorphic terms and should not cause any confusion. When talking about classical agent theories the intentional stance is both useful and harmless as a shorthand to describe behavior, but describing the behavior of LLMs should require a bit more thought.

2.2 The data to be analyzed

The result of each test case is reported via an Android app and the result is currently written to an Excel sheet that is openly available within the organization, to facilitate the flow of information to stakeholders. This Excel sheet is called “Gonogo” and contains the information necessary for decisions relating to whether to “go” or “not go” to the next phase of development with the truck and allow this truck to be driven on open roads.

The sheet contains almost 50 columns and tens of thousands of test cases at any given time. Many columns are redundant to an experienced verification engineer or just informative and LLMs have a tendency to confuse column names (see Section 2.3). As such, a smaller set of default columns was agreed upon together with a domain expert, which are selected when reading the file into the agent. Each test case relates to a specific part (function) of a end-user function (EUF) for each truck variant that is verified. For example, “max defrost” is a function within the EUF “Cab Climate”. The same function is tested within many different truck variants.

[1]The further from humans in likeness a subject is, the weaker these networks will fire. For example, we recognize actions taken by chimpanzees easily and can relate, but we have a harder time relating to insects.

2.3 Expected analysis tasks

Two common questions from stakeholders are **A:** "What is the truck status of X?" and **B:** "What is the Y-status of X?", where X is the ID of a specific truck and Y is an EUF. Some knowledge of the domain and the data is required to extract the answers and communicate them back to the stakeholder. For example, to answer question **A** an analyst needs to know that "truck status" refers to the EUFs that failed any test in the latest test-suite (which in turn corresponds to the highest number in the column `UniqueID`). To answer question **B** the analyst needs to know that Y corresponds to a value in the `euf` column, and to report the counts of the values in the `result_status` column.

Some questions require solutions that are more complex than others. To answer question **A** above, only *filtering* operations are necessary, while question **B** also requires *counts* of passed and failed tests for the specified EUF. Some questions are more general, such as **C:** "What EUF has the highest proportion of failed test cases?". Such questions require grouping the data in addition to filtering and counting operations, and the operations need to be done in the correct sequential order, raising the overall complexity of the task. The concept of task complexity is covered in more detail in Section 3.6

The tasks for the agent are expected to be defined and queried by a verification engineer or someone else who has the relevant domain knowledge, to translate any questions to a more precise query that declares exactly what information is requested from the data. A query to an LLM needs to be specific to minimize the opportunities for hallucinations, Pourreza and Rafiei (2023) report that the cause of 37% of their failed queries is because of schema-linking errors. Dong et al. (2023) report that LLMs tend to include redundant columns unless instructed not to, adding the text "do not select extra columns that are not explicitly requested in the query" to their prompt.

2.4 Practical considerations

One of the most pervasive tools for tabular data analysis, excluding spreadsheet-applications such as Excel, is the library pandas (McKinney, 2010; pandas development team, 2024) for the programming language Python [2]. Since many LLMs are proficient both at Python generally and pandas specifically, the choice to perform the analysis tasks by executing Python code falls naturally and is in line with state-of-the-art LLM-agents (e.g., Hong et al., 2024; Qiao et al., 2023).

In any industrial setting it is likely that only a few LLMs, or none, are served, and the price of using the most recent proprietary models are orders of magnitude more expensive than older models. As such the agent cannot be too reliant on the capacities of the latest and greatest, the agent mechanisms and prompts must cater to less capable LLMs. In developing the agent we had access to `CodeLlama 3 70B` [3],

[2] https://www.python.org

[3] https://ai.meta.com/blog/code-llama-large-language-model-coding/

`Mixtral 7x8b` [4], and `GPT-3.5`.

We use the `Lagent`-framework (Team, 2023) to build our agent. As previously mentioned in Section 1, some existing agent frameworks are much too complex or has a very large code base, which is a concern for the team we are collaborating with. As such, the team had already adopted an early version of Lagent. This choice aligns well with our project since existing LLM-agent frameworks tend to be overly general or complex for our specific needs. `Lagent` is presented in more detail in Section 3.5.

[4] https://mistral.ai/news/mixtral-of-experts/

3

Theory

This chapter begins by presenting the field of autonomous intelligent agents, highlighting some challenges in classical theories. The next section covers LLMs, their architecture, and techniques that are relevant for using LLMs to solve problems. The section after covers research on LLM agents: agent systems that incorporate LLMs as solutions to long-standing classical agent theory challenges. Finally, to facilitate the evaluation of our agent, we present theory on general task complexity and how it can be applied to tabular data analysis tasks performed by using Python.

3.1 Intelligent agent theory

Research on autonomous intelligent agents has been heavily intertwined with, and sprung from, cognitive science (Dyachenko et al., 2018; Varela et al., 1991). Agent capacities are generally modeled from biological cognitive systems. Agents' decision making capacity have either been programmed in a deliberate way or learned through RL, making generalization hard. Also, how to interact with agents has been a well-researched problem, i.e., how they understand and follow instructions and how to encode, store, and retrieve knowledge (Russell & Norvig, 2021).

Defining what an agent actually is has proven to be difficult, since what constitutes an agent greatly differs depending on context and application (Franklin & Graesser, 1996; Luck & d'Inverno, 2001; Wooldridge & Jennings, 1995). A general definition of an agent comes from Russell and Norvig (2021): "*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators,*" and one of the more cited definitions of an *autonomous* agent comes from Franklin and Graesser (1996): "*a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future*". The definition is explained further by comparing systems on the high end of being an agent (humans and animals having many senses and actions) to systems on the low end (a thermostat having one or two senses and one or two actions). However, it is generally recognized that there really is no agreement on what constitutes an agent and as such most research provides their own definition (Luck & d'Inverno, 2001).

In lack of consensus definitions, agents are more often described and defined in relation to what they are expected to do and how they do it. A *cleaning agent*, a *software agent*, and a *robot assistant* are all agents but share only few properties.

Autonomous mobility would be a defining characteristic of any robot agent but not applicable to the other two (except perhaps certain types of software agents that autonomously copy themselves to other systems, i.e., a computer virus). In addition to this implicit requirement of a goal or purpose, general characteristics of agents can be condensed to (Cheng et al., 2024):

- **Autonomy**: One major gain from developing an agent is to reduce human intervention, the agent needs to independently perform its task.
- **Perception**: In order to perform any task independently it needs to be able to perceive relevant information in its environment.
- **Decision-making**: Based on its perception the agent needs to decide on an action to take.
- **Action**: Anything that alters the state of the environment.

These characteristics have historically been implemented through different kinds of agent programs, categorized in Russell and Norvig (2021) as *simple reflex agents, model-based reflex agents, goal-based agents*, and *utility-based agents*. Each of these program types can also be turned into a *learning* agent, an agent that improves its performance over time based on its own percepts, actions and feedback from the environment or an internal critic module. The programs are listed in order of increasing complexity, where simple reflex agents take a decision solely based on the current percept and a set of condition-action rules. A model-based agent will maintain a model of the world based on its percept history and match a condition-action rule based on the current state. A goal-based agent is a model-based reflex agent that chooses an action based on its world model and goal state. A utility-based agent further involves a utility function that can take more factors into account rather than just goal fulfillment or failure.

These agent programs have been successfully implemented on many specific tasks. For example robot lawnmowers and robot vacuum cleaners are mostly examples of utility-based agents that take into account not only level of task completion when choosing an action, but also battery-level and perhaps even current weather conditions (to avoid cutting the grass when it is wet).

Applying these concepts on our automation problem (Figure 1.1) and ignoring the technical *how* of each part, we would have a goal-based agent that has the goal of completing a given analysis task. It would need to understand an analysis problem stated in natural language and set the goal to deliver data that solves the problem. To translate the problem to a series of actions it would need to *plan* the actions necessary to complete the analysis task, which requires knowledge about what actions it *can* take and how those actions interact with each other. It would then need some *actuator*, some mechanism with which to act on the environment.

3.1.1 Agent engineering challenges

There are classical theories on how to make agents reason about a task, how to implement an understanding of logic, and how to plan (e.g., forward or backward search, or hierarchical plans)(Russell & Norvig, 2021). Somewhat successfully on

easy tasks, but carries immense costs in designing and searching larger problem- and action spaces.

Decision-making is often realized with a combination of condition-action statements based on percepts, but reinforcement learning and evolutionary algorithms have also been widely employed to learn successful behaviors (Russell & Norvig, 2021). None of these methods allow for generalization to new domains, and except for some evolutionary paradigms they do not allow for any greater degree of adaptation and variability - leading to major problems when unforeseen tasks or difficulties arise.

Other challenges in classical agent theories are *knowledge representation* and *communication*. For an agent to follow instructions it must understand those instructions, and for an agent to make sound decisions it needs knowledge about its own capacities and the environment it acts in, and even the relationship between its own actions and how they change the environment. Classical theories of knowledge representation rely heavily on hand-crafted ontologies and logic, which has given rise to *ontology engineering*. It is no easy task, as engineering an ontology requires formal definitions of categories, objects and their properties, events, as well as relationships between entities. The simplest ontologies apply first-order logic but it quickly becomes apparent that it is not sufficient for representing *real world* knowledge. Most real world knowledge has exceptions or holds only under certain circumstances or to a certain degree, which introduces the need for more complex models such as *semantic networks* with *modal logic* or *description logics* (Russell & Norvig, 2021).

Communication has historically relied on some *agent communication language* for message passing and input/output to the agent (e.g., Finin et al., 1994), as well as other techniques such as pattern matching, semantic grammar, or syntax-based systems (Androutsopoulos et al., 1995).

To summarize, agent theories' biggest challenges has been how to make agents understand and follow instructions, how to represent and use knowledge, and how to reason and plan a sequence of actions towards a goal. Symbolic problem spaces explode with these challenges and require immense manual labor. However, the advent of large pre-trained language models infused the field of intelligent agents with new vigor, identifying LLMs as a potential agent "brain" that not only can understand and produce real natural language, but also seems able to plan actions towards a goal and reason about that plan.

3.2 Understanding the question

The task of understanding a natural language question or instruction is referred to as NLU and is a type of task (A. Wang et al., 2018) addressed within the long-standing research field of NLP (Joseph et al., 2016), with historic progress following the general arch of breakthroughs in NLP-technology. From *symbolic* and *statistical* methods to the successful application of *connectionist* methods (Johri et al., 2021; Liddy, 2001). Parsing the input symbolically requires very strict rules for the format and contents of the query, and mapping the parsed input to actions requires a lot of manual engineering of conditional rules as well as ontologies of attributes,

relationships, and possible operations. Statistical methods perform better on many NLP-tasks but has not been widely leveraged as a tool in agent research, nor in other fields that directly relate to producing actions for data analysis (e.g., Popescu et al., 2003; Russell & Norvig, 2021).

Beginning around 2010, the connectionist approach has been able to bring breakthrough after breakthrough in NLP, enabled by the same factors that are fueling the "third AI-boom" (Miyazaki and Sato, 2018; Nakabayashi and Wada, 2016) - namely *big data* and the will and resources to process and analyze it using machine learning. As data availability and computing power increased, *deep* neural networks (LeCun et al., 2015) based on recurrent neural network (RNN) and long short term memory (LSTM)-architectures (Hochreiter & Schmidhuber, 1997) were finally able to better capture longer-distance dependencies between words and semantics from sequential information. Google made the switch to neural networks for machine translation in 2016 (Y. Wu et al., 2016), employing a sequence-to-sequence (Sutskever et al., 2014) architecture of stacked LSTMs and an attention module (Bahdanau et al., 2014). Zhong et al. (2017) applied the sequence-to-sequence architecture on translating a question to a structured SQL query using RL.

3.2.1 Large language models (LLMs) understand natural language

The most recent breakthrough, that very few have missed, is the advent of pre-trained LLMs. Most famously, it is the GPT-family (Radford & Narasimhan, 2018) of LLMs that enable the capabilities of OpenAI's commercial ChatGPT-product[1]. LLMs are deep artificial neural networks based on the Transformer architecture (Vaswani et al., 2017), pre-trained on massive text datasets compiled mostly from crawling the internet (Y. Liu et al., 2024). Being a language model, it models the conditional probability of the next token[2] given all the preceding tokens in a sequence (Bengio et al., 2000):

$$P(x_1^n) = \prod_{i=1}^{n} P(x_i \mid x_1^{n-1}),$$

where x_1^{n-1} represents the sequence of tokens $\{x_1, x_2, ..., x_{n-1}\}$.

This enables the model with parameters θ (p_θ) to sample the next token y from the joint probabilities over the vocabulary given the sequence of tokens that precede the prediction:

$$\hat{y} \sim p_\theta(x_{n+1}|x_1^n),$$

where x_1^n is the preceding sequence of tokens.

[1] https://chat.openai.com

[2] The words in the input sequence are broken up into sub-word pieces called tokens via a learned *tokenizer*. The text input to a pre-trained LLM must pass through the same tokenizer that the LLM was trained with, since its weights are adapted to those specific tokens and those specific tokens make up the model's vocabulary.

Inner workings

The breakthrough research that gave rise to LLMs comes from Vaswani et al. (2017), a now famous paper[3] called "Attention is all you need". This paper introduces the Transformer architecture.

Attention-mechanisms (Bahdanau et al., 2014; Luong et al., 2015) had at the time been successful in improving long-distance-dependency detection in RNN- and LSTM (Hochreiter & Schmidhuber, 1997) architectures, and Vaswani et al. (2017) were first to apply attention in a novel non-recurrent architecture. Instead of sequentially operating on a sequence of tokens, the attention mechanism allows for computing dependencies and relevance for all tokens in the input sequence in relation to each other, getting rid of the need for both recurrence and convolutions. This makes the model much less computationally expensive and able to train on much more data.

The Transformer architecture consists of a series of encoder and decoder blocks (Figure 3.1), where the encoders apply multi-head (bi-directional) attention on the entire input sequence and passes the representation through feed-forward multilayer perceptron (MLP) and normalization layers in each block. This encoder-decoder architecture is very performant on sequence-to-sequence tasks, such as machine translation, where bi-directional context is very relevant. However, LLMs generally only employ the decoder block in order to be auto-regressive language generators, to model the next token probability based only on the preceding tokens. Instead of normal multi-head attention the decoder first applies a *masked* attention layer, that for every token in the input sequence excludes all subsequent tokens from the attention computations.

Transformer-based language models have shown a remarkable capacity to solve NLP tasks such as NLU (Z. Yang et al., 2019), information extraction and named entity recognition (Dunn et al., 2022), and summarization (Raffel et al., 2020), promising progress wherever NLP-issues before was a bottleneck.

3.3 Translating the question to actions

In an agent framework the translation to actions would correspond to a *planning* stage or module. This has also historically been approached with symbolic methods, with conditional statements, planning algorithms, and ontologies describing the environment and agent capacities (Russell & Norvig, 2021). This is not an easy problem, but is also addressed by the capacities of LLMs.

A side effect of next-token-prediction given massive pre-training, is that the generated text sequences can showcase vast general and specific knowledge about the world (Petroni et al., 2019; Roberts et al., 2020). See Listing 2 in Appendix for an example of a short dialogue with ChatGPT that illustrates the knowledge its lan-

[3]To emphasize the impact of this research: Vaswani et al. (2017) has come to be one of the most cited papers of all time with 119 393 citations according to Google Scholar at the time of writing, surpassing number four on Natures list from 2014 with the 100 most-cited research papers of all time (Van Noorden et al., 2014).

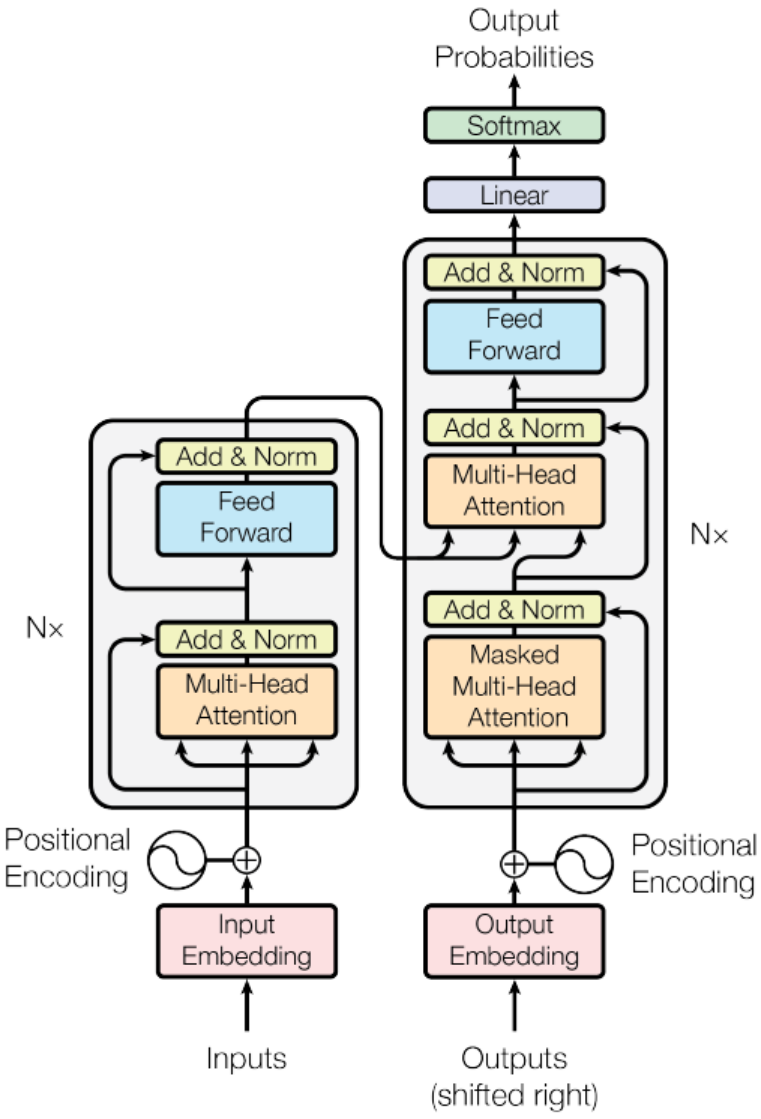


Figure 3.1: The Transformer architecture. Figure from Vaswani et al. (2017).

guage carries about tabular data analysis. The most robust knowledge comes from concepts that were more prevalent in the training data (Kandpal et al., 2023), and concepts that were less prevalent are much more prone to *hallucination*, meaning that the model is more likely to output "content that is nonsensical or unfaithful to the provided source content" (Ji et al., 2023, p. 248:3) in relation to such lesser known concepts.

Another effect comes from the fact that the internet is full of examples of source code in many different programming languages, often in close proximity to descriptions of what the code does (e.g., websites like StackOverflow [4] and GitHub [5]). This means that most pre-trained LLMs are able to generate code quite proficiently (Austin et al., 2021; Xu et al., 2022), at least for the most popular languages (e.g., Python, SQL).

As such, LLMs have knowledge about data analysis procedures to help them plan analysis tasks, and they have a potent actuator with which to act on the data. However, just asking ChatGPT to plan a multi-step task often results in hallucinations (Valmeekam et al., 2023), and asking it to generate an entire solution in code at once rarely results in executable code (Chen et al., 2023). There are, however, numerous techniques that enhance the planning and code generation capabilities of LLMs.

3.3.1 Solving complex problems with LLMs

In addition to pure NLP-tasks, LLMs have been leveraged to solve more complex tasks requiring planning and reasoning (Huang & Chang, 2022). For example, just like for us humans (Y. Wang & Chiew, 2010), especially in dynamic contexts where one part of the solution depends on another, it can help to decompose the problem into smaller sub-problems (Pourreza & Rafiei, 2023). Zhou et al. (2022) decompose a question by prompting an LLM to decompose it into sub-questions, increasing accuracy on compositional generalization benchmark SCAN [6], and Khot et al. (2022) decompose tasks into specific types of sub-tasks that are passed to specific task-handlers.

One remarkable aspect of LLMs is that they can learn new tasks just by having examples of similar tasks being solved present in the input text (*few-shot learning*, Brown et al., 2020), and their performance can increase when prompted to reason (*chain-of-thought (CoT) prompting*, Wei, Wang, et al., 2022) or think step-by-step (*zero-shot CoT*, Kojima et al., 2022). These capacities can again make LLMs feel very human-like and it can be easy to think that there must be something more to it than just next-token prediction going on. But as Murray Shanahan, cognitive scientist and senior researcher at Google DeepMind, states in Shanahan (2024, p.77):

> Well, it is an engineering fact that this is what an LLM does [next-token prediction]. The noteworthy thing is that next-token prediction

[4] https://stackoverflow.com/
[5] https://github.com/
[6] https://github.com/brendenlake/SCAN

> is sufficient for solving previously unseen reasoning problems, even if unreliably. How is this possible? Certainly, it would not be possible if the LLM were doing nothing more than cutting-and-pasting fragments of text from its training set and assembling them into a response. But this is not what an LLM does. Rather, an LLM models a distribution that is unimaginably complex and allows users and applications to sample from that distribution.

However, instructing an LLM to write a program or a complex SQL-query rarely results in correct code on the first generation attempt. Also, while programming is a powerful tool when performing actions and solving problems, the model itself only outputs text. That text needs to be handled and executed by mechanisms outside the LLM.

3.4 Executing the actions

LLMs are impressive on their own but also quite useless if your goal is to automate tasks, since all they do is predict text tokens. To leverage the capabilities of LLMs in an agent framework we need to provide it with perception, actions, and an environment to perceive and act in.

In classic agent theory, agents perform actions in their environment through actuators and perceive their environment through sensors. Robots have physical *effectors* (e.g., arms, grippers, motors) and sensors (e.g., camera, range detector, battery level) and perceive and act on physical environments. Software agents act in a virtual environment, acting through functions (e.g., write to disk, press button) and perceiving its environment through virtual sensors (e.g., file contents, screenshots) as well as physical sensors (e.g., keyboard, touchscreen, microphone). (Russell & Norvig, 2021) Historically the interaction between perception and action has been programmed symbolically or learned through RL, again, making agents difficult to build and hard to generalize.

The very earliest agent based on LLM technology was `WebGPT` (Nakano et al., 2021), which showcased an agent that given a question searches the web and delivers an answer. A closed environment was created with a text based web browser which the agent learned to use through *imitation learning*. Imitation learning is a paradigm of RL where the reward function is based on the performance compared to expert human demonstrations. The development included substantial data gathering from human experts and fine-tuning GPT3-models in four different ways, creating an agent that is apt at a very specific task. This early concept was able to leverage the capabilities and knowledge of GPT-3, but still relied on high-resource methods to acquire its web-search capability. WebGPT perceived its environment through a text-representation of the state of the web-browser, and generated navigation commands became actuators that searched and scrolled for answers.

With the advent of chain-of-thought (CoT) (Wei, Wang, et al., 2022) and the ensuing paradigm of *pretrain, prompt, predict* (P. Liu et al., 2021), and as models became larger and more capable (Chowdhery et al., 2023; Rae et al., 2021), came

```
Question: Author David Chanoff has collaborated with a U.S. Navy admiral
who served as the ambassador to the United Kingdom under which President?
Thought 1: I need to search David Chanoff and find the U.S. Navy admiral
he collaborated with.
Action 1: Search[David Chanoff]
Observation 1: David Chanoff is a noted author...
Action 2: Search[William J. Crowe]
Observation 2: William J. Crowe is...
Thought 3: William J. Crowe served as the ambassador to the United Kingdom
under President Bill Clinton. So the answer is Bill Clinton.
Action 3: Finish[Bill Clinton]
Observation 3: Episode finished
```

Listing 1: Example of a ReAct reasoning trace. Reproduced from the appendix in Yao et al. (2022). Each action is parsed and performed by engineered agent mechanisms and the results are included in the input for generating the corresponding observation.

more general agents that relied more on *prompt-* or *mechanism engineering* rather than fine-tuning and RL (Z. Zhang et al., 2023). Mechanism engineering refers to engineering mechanisms in the agent code to facilitate agent capability acquisition, for example providing mechanisms for *trial-and-error*, *crowd-sourcing* (among many LLMs), or *experience accumulation* (for examples of each see L. Wang et al., 2023).

Based on the idea from CoT that LLMs learn to reason from in-context examples, Yao et al. (2022) proposes a framework that allows an LLM to perform actions (by specifying that action as a response) and reason about its observations. The idea behind `ReAct` is simple and effective. By prompting the LLM with few-shot in-context examples of human made reasoning traces, the LLM will itself produce a reasoning trace that aids it in solving the task. The reasoning trace consists of an interleaved sequence of *thoughts, actions,* and *observations*, see Listing 1 for an example. The in-context examples and the action space can be modified for different tasks and domains, making the agent flexible. The capability of `ReAct` to solve complex problems with large actions spaces is however constrained by the size of the context window of the LLM, as more demonstrations are needed and longer reasoning traces are generated.

Agent self reflection as an iterative feedback loop

As mentioned in Sections 2.4 and 3.3, an LLM's ability to generate Python code can be leveraged as an agent actuator with which it can perform data analysis actions. In practice it means building a trial-and-error mechanism that pass generated code as input to a module that can execute the code and return any result produced, along with potential warnings or errors. The agent can then reflect on the results and decide on the next step.

`TaskWeaver` (Qiao et al., 2023) implements this iterative feedback in a code interpreter (CI) module, see Figure 3.2. The CI receives a query from the `Planner` module and generates code that should answer the query. The generated code is

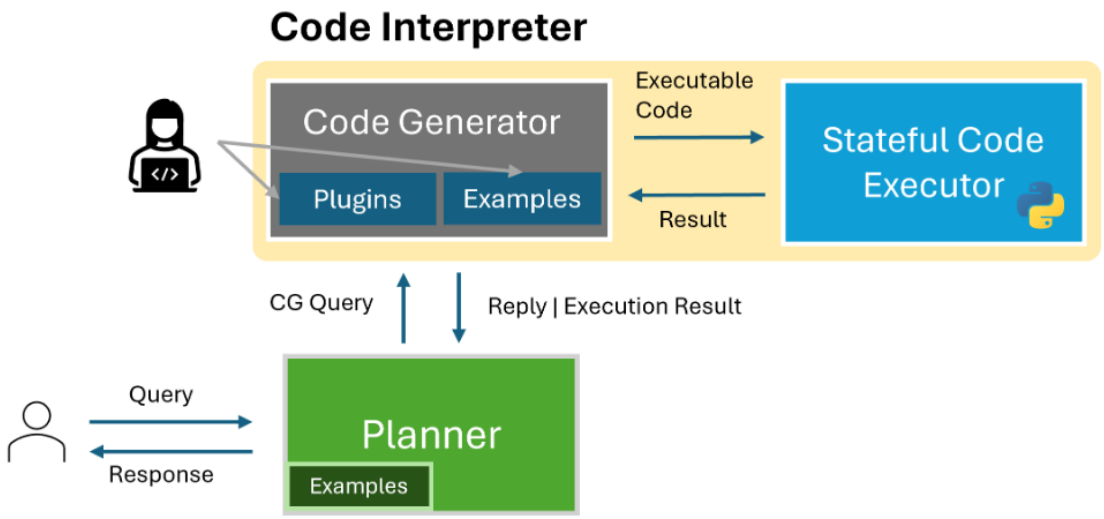


Figure 3.2: Overview of TaskWeaver. Figure from Qiao et al. (2023).

then executed and any results are passed from the execution environment back to the LLM that generated the code, the `Code Generator`. If the code fails to run the `Code Generator` will try to re-generate code until the code executes without errors or until a set maximum number of tries is reached, upon which the failure is reported back to the `Planner`. If the code executes without errors, the result is passed directly back to the `Planner`.

When researching LLM-agents for this thesis we identified this *iterative feedback approach* as a general agent process that would be suitable for an agent performing data analysis with Python code, since analysis tasks consist of steps that often depend on certain results from previous steps. The iterative feedback approach is the basic trial-and-error mechanism that lets the agent perform an action and lets an LLM observe the result of the action before deciding on the next action, like the thought-action-observation loop in `ReAct` (Listing 1). A description of the iterative feedback approach implemented for this thesis is presented in Section 4.2.4 and Figure 4.1. This mechanism for specifically coding was later labeled as the "interpreter paradigm" by Hong et al. (2024).

3.5 Related work: Intelligent LLM-agents for data analysis

TaskWeaver is a *code-first* agent, meaning that it always converts a user query to executable Python code. It is a general agent in the sense that it can be adapted to specific tasks by being provided task-specific examples and functions. No training of any model weights has been performed or is needed. TaskWeaver is a general agent while the agent developed in this thesis is specifically for answering questions about the Gonogo dataset (Section 2.2). Implementing TaskWeaver in the industry setting in this thesis would bring unwanted complexity and unnecessary features,

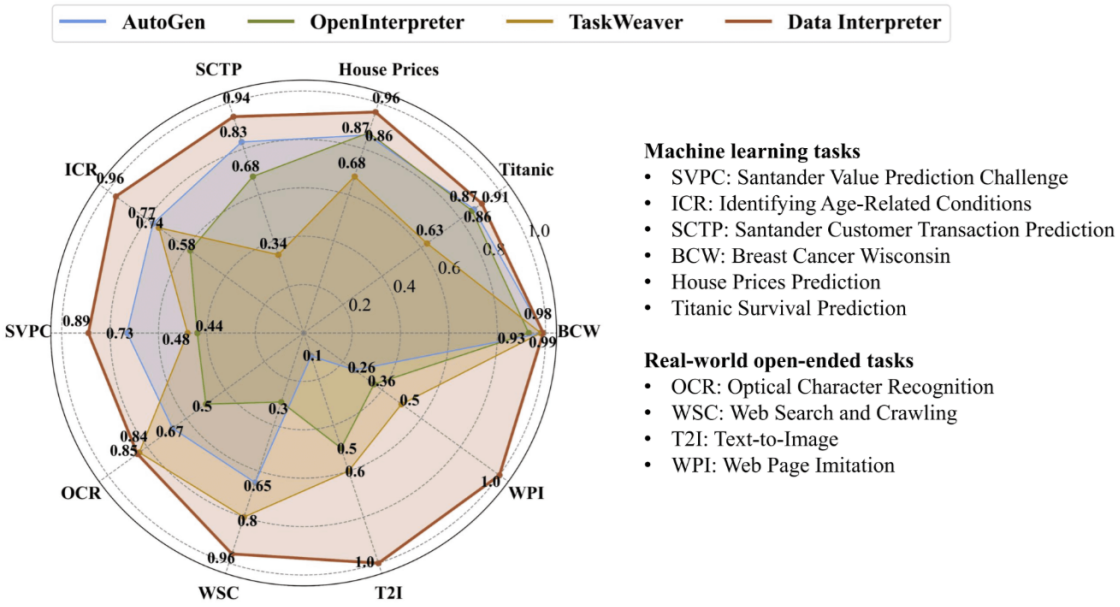


Figure 3.3: Data Interpreter outperforms other interpreter paradigm agents on specific machine learning tasks and real-world open-ended tasks. Figure from Hong et al. (2024).

as well as a completely new codebase to the in-house agent repository. Our initial experiments with TaskWeaver with the `Phind CodeLlama 34b` model also revealed issues for this LLM to generate responses in accordance with the strict structure defined by TaskWeaver prompts. If properly configured, TaskWeaver should be able to perform tabular data analysis tasks.

Data Interpreter (Hong et al., 2024) is an LLM agent for data science, able to autonomously solve classic challenges (e.g., House prices, Titanic survival prediction) as well as more open-ended tasks such as *web search and crawling* and *optical character recognition* (OCR). Data Interpreter is the current state-of-the-art agent regarding data analysis and can do much more than tabular data analysis. Figure 3.3 shows how it outperforms other agents in the interpreter paradigm. The other agents are `AutoGen` (Q. Wu et al., 2023), `OpenInterpreter` [7], and `TaskWeaver`.

GPT4 Code Interpreter (OpenAI, 2024) is a commercial product from OpenAI that provides a sandbox environment for Assistants (another OpenAI product) to execute code in. Using the Code Interpreter comes at a prize at the time of writing of $0.03 per session. The authors of this thesis have not interacted with this product, but the knowledge of it existing inspired some initial ideas.

Lagent (Team, 2023) is a light-weight framework that enable users to quickly experiment with different types of agents. The framework is designed around an *action executor* module, a LLM, and a *planning and action* step. Implementing different types of agents, as in defining how the planning and action happens, is controlled by defining a protocol and an agent class. As mentioned in Section 2.4, version

[7] https://github.com/OpenInterpreter/open-interpreter

0.1.2 of this framework was used to provide a very basic skeleton for classes and data structures for agents at Volvo. The agent developed during this thesis uses the same version as a foundation, with the addition of an *IPython interpreter* from Lagent version 0.2 implemented as an agent action. The Ipython interpreter provides a stateful environment in which to execute blocks of Python code, meaning that variables persist between execution rounds. Section 4.2.2 describes the Code Interpreter module developed during this thesis.

3.5.1 Answering research question 1

Many agents use a general feedback loop where an action is followed by an LLM observing the result and then deciding on the next action (e.g., Dasgupta et al., 2023; Hong et al., 2024; Shinn et al., 2023; Q. Wu et al., 2023; Yao et al., 2022). Both `TaskWeaver` and `Data Interpreter` employ a `Code Interpreter` module which generates and executes code iteratively in a stateful `IPython` environment. Both agent frameworks also employ a module that decomposes the incoming query into a plan, from which each step of the plan is passed to the `Code Interpreter`, and the result is then reflected upon by the planning module before the next step is either revised or directly passed to the `Code Interpreter`.

Based on our literature review, the most suitable agent architecture for tabular data analysis is an architecture with 1) a module for decomposing and planning the task, 2) a module for code generation and execution, and 3) a loop in where the planning module passes one step at the time to the coding module and reflects on the results before revising the next step or passing it as-it-is for code generation. Our implementation of this iterative feedback approach is described in Section 4.2.4.

3.6 Task complexity

In evaluating an agent we want a measure of task complexity to understand its capacities and limits. For an agent to be useful it must be able to solve tasks with a certain minimum complexity, and each different agent will most likely have a ceiling of complexity where it no longer is able to arrive at a correct solution. We propose a metric to compute the complexity of a tabular data analysis task using Python, and relate that metric to the agent's rate of successful task completions. We are working under the assumption that there is a ceiling of complexity where certain agent configurations no longer solve any analysis task.

Measuring task complexity is a difficult problem where recent research proposes very complex models that are entangled with the task performer and context (Danner-Schröder & Ostermann, 2022; Haerem et al., 2015). Inspired by Topi et al. (2005), we apply the original construct[8] from Wood (1986) on our domain. See Section 5.2 for our application, below follows a summary of the relevant concepts from the original research.

[8]The construct from Wood (1986) is still widely used in task complexity research today (Almaatouq et al., 2021; P. Liu & Li, 2012).

Wood (1986) defines the construct of task complexity as a linear relationship between three types of complexity: *component complexity*, *coordinative complexity*, and *dynamic complexity*. These types of complexity are based on measures of the three constructs that describe any task: *acts*, *information cues*, and *products*.

Products are what is created or produced by a sequence of *acts*, and a product can always be described without referring to the acts that produce them. Acts are defined as "patterns of behaviors with some identifiable purpose or direction" (Wood, 1986, p. 65) and is exemplified with the act of "lifting", in contrast to a very specific activity like "clasping fingers" that does not serve any additional purpose and cannot really be described without referring to the act of clasping fingers. *Information cues* are elements of information required for the task.

Component complexity refers to the number of distinct acts required for successful task completion and the number of distinct information cues that need to be handled in the performed acts. Component complexity can increase further by including subtasks as input, each with its own acts and information cues. A general formula for component complexity of a task is as such

$$TC_l = \sum_{j=0}^{p} \sum_{i=1}^{n} W_{ij} \tag{3.1}$$

where n = number of distinct acts in subtask j, W_{ij} = number of information cues that require processing in the i^{th} act of the j^{th} subtask, p = number of subtasks, and TC_1 = component complexity.

Coordinative complexity refers to precedence relations among required acts, where coordinative complexity increases with the length of the sequence of such dependencies. Coordinative complexity could also include factors such as timing, frequency, and intensity requirements for acts, where more such requirements will demand more knowledge and skill of the agent performing the task. A formula for capturing only the precedence dependencies of a task is

$$TC_2 = \sum_{i=1}^{n} r_i \tag{3.2}$$

where n = number of acts in the task, r_i = number of precedence relations between the i^{th} act and all other acts in the task, and TC_2 = coordinative complexity.

Dynamic complexity refers to changes in component and coordinative complexity of the task, over time. A formula for the differences is given by

$$TC_3 = \sum_{f=1}^{m} \left| TC_{1(f+1)} - TC_{1(f)} \right| + \left| TC_{2(f+1)} - TC_{2(f)} \right| \tag{3.3}$$

where TC_1 = component complexity, TC_2 = coordinative complexity, f = the number of time periods over which the task is measured, and TC_3 = dynamic complexity.

Query	Component	Coordinative	Total
Return failed tests for truck [Truck ID].	3	0	3
For each release candidate, list the test case functions that fail the most.	15	6	21

Table 3.1: Examples of one simple query and one complex, with complexity calculations based on the gold-standard solution.

Total task complexity can as such be expressed as a linear relationship between the aforementioned types of complexity:

$$TC_t = \alpha TC_1 + \beta TC_2 + \gamma TC_3 \tag{3.4}$$

where $\alpha > \beta > \gamma$.

Applying this to tabular data analysis we can exemplify one query that has low complexity and one that has high complexity, see Table 3.1. We do not expect dynamic complexity to have major relevance in the domain of tabular data analysis, since every task is quite well-defined and neither the required acts nor the sequence they are performed in are expected to change at all.

3.7 Evaluating table analyst LLM-agents

LLMs are usually evaluated on a number of different tasks and benchmarks, to cover different aspects of their capacities. Chang et al. (2023) outlines *what* aspects of an LLM to evaluate, *where* to evaluate (i.e., datasets and benchmarks), and *how* to evaluate (i.e., automatic or human evaluation). What to evaluate includes, of course, NLP-capacities but also dimensions such as *robustness*, *ethics*, *biases*, and *trustworthiness*, as well as how well the LLM performs in specific applications.

Where to evaluate is divided into benchmarks for general tasks and benchmarks for specific downstream tasks. Benchmarks for general tasks evaluate things such as proficiency in multi-turn dialogue (MT-Bench, Zheng et al., 2023), proficiency in evaluating quality of LLM output (LLMEval, X. Zhang et al., 2023), or they measure some subjective measure from human interaction (Chatbot Arena, Chiang et al., 2024). Holistic benchmarks such as HELM (Liang et al., 2022) tries to aggregate evaluations across many aspects of LLM-technology, while BIG-bench (Beyond the Imitation Game benchmark) (Srivastava et al., 2022) is a benchmark that focuses on tasks that are believed to go beyond the current capabilities and as such is extremely challenging for current LLMs.

Specifically for NLP-tasks, the most popular benchmarks are GLUE (A. Wang et al., 2018) and variations of it, e.g., SuperGLUE (A. Wang et al., 2019) and X-GLUE (L. Yang et al., 2022). GLUE is a collection of NLU-tasks including question-answering, sentiment analysis, paraphrasing, and natural language inference-tasks.

To the best of our knowledge there are currently no public benchmarks that can evaluate the agent proposed in this thesis. One related work is WikiSQL, which is a

benchmark created by Zhong et al. (2017) to evaluate text-to-SQL generation. Another work is DS-1000 (Lai et al., 2023) which is a "natural and reliable benchmark for data science code generation". This work evaluates the proficiency to solve problems with Python libraries such as `Pandas`, `numpy`, `PyTorch`, `Scikit-learn`, and `Matplotlib`. The problems are general and often toy-examples where the construction of dataframes are presented in the input to the model, it does not evaluate how well an agent would perform when answering questions about the Gonogo datasheet.

To facilitate automatic evaluation of our agent we develop a *dataframe similarity* metric (see Section 5.3) and a small novel dataset from the original real-world use-case questions we received from a domain expert. We follow standard practice in benchmark literature (Chang et al., 2023) and provide gold-standard solutions to each query, see Section 4.1 for more details.

The most important metric in evaluating our agent is how useful it is as a tool for Thomas the Verification Engineer, a metric that can only be gathered subjectively. As such we provide a domain expert with solutions generated by the agent and survey qualitatively their evaluation of the solutions and correlate their score to our dataframe similarity score. The domain expert evaluation is described in Section 5.5.

4

Design and Implementation

4.1 Dataset and Use-Cases

Initially, we received 15 analysis tasks (use-cases) to test our agent. However, many of these use-cases contained ambiguous requests. For instance, terms like *truck status* in the use-case "What is the truck status of [Truck ID]?" or *current issues* in the use-case "What are the current issues on [Truck ID]?" were unclear. To address this, we collaborated with a domain expert, a test verification engineer, to clarify these use-cases. We then translated them into explicit table analysis queries that our agent could process.

We needed a way to evaluate our agent's performance objectively. This requirement led us to develop solutions for all our table analysis queries in `Python`. We compared and discussed these solutions to agree on a definitive ground-truth solution.

Upon reviewing our ground-truth solutions, we observed that these `Python` code solutions could be neatly broken down into smaller code chunks. We identified operations such as filtering, column selection, and sorting that could be separated.

Running our prototype agent on the initial tasks, we noticed that our agent struggles with most of them. This led us to defining a hypothesis:

Hypothesis 1 *Our LLM agent can handle analysis tasks under some threshold of complexity.*

Motivated by our Hypothesis 1 and an assumption that longer tasks are more complex, we investigated the relationship between query complexity and the number of steps or code chunks in our solutions. In other words, we wanted to see if longer queries with more steps were indeed more complex for the agent to handle. To explore this, we created ablations of our queries.

4.1.1 Query Ablations

We began creating our query ablations for each query by taking the first code chunk from our solutions and writing a query that this code chunk would solve. For example, if the first code chunk in our solution filtered "NOK" tests, our first ablation query was "Return all failed tests."

We then cumulatively added the next code chunk to our ablation and wrote a query

that both code chunks would solve. For instance, if we added a code chunk that returns counts of rows for each truck (`df.value_counts("name")`), our second ablation query was "Return the counts of failed tests for each truck". We expected the agent to understand that it needed to first filter "NOK" tests and then count the number of rows for each truck.

We continued this process, adding additional code chunks and generating a query for each ablation until we reached the full code. This approach expanded our initial 15 table analysis queries into 50 query ablations, which also include the original 15 queries. Each query and query ablation had a corresponding ground-truth solution and `Python` code to achieve this solution.

4.2 Agent design

In this section, we describe the design of our table analyst LLM agent. This includes the LLM-powered modules, the communication approach between them, and the prompting strategy we use.

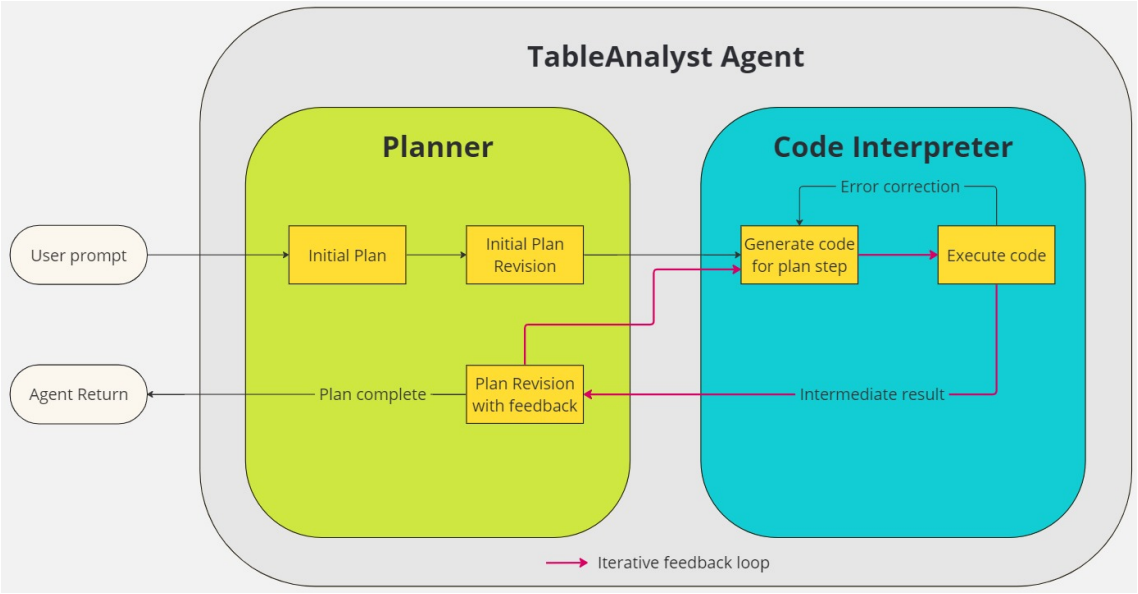


Figure 4.1: TableAnalyst Agent Diagram

4.2.1 Planner Module

The planner is a separate, self-contained module within our LLM-powered agent. It acts as the agent's brain, generating and revising step-by-step plans to solve the user's query task. It houses all methods related to plan creation and revision and stores all plans and revisions internally.

The plans are designed such that each step is simple enough for the CI (covered in section 4.2.2) to execute independently. This means each step should be self-contained, with no references to actions outside of the current step. For example, if

the previous step asked to group the dataframe and calculate means stored in a new column named "mean_values", the next step should explicitly refer to this value by name, such as "sum up mean_values" instead of "sum up the mean calculated for each group".

4.2.2 Code Interpreter Module

The CI's purpose is to generate, execute, handle errors, and correct code that solves a single step in the plan generated by the planner. Because of this singular focus, the CI performs no reasoning; all reasoning is handled by the planner.

Our agent allows for stateful code execution. This means we initialize a Jupyter kernel with a persistent environment, where global variables are accessible across different code execution sessions. This allows us to store intermediate results as pandas dataframes in the global environment, which can then be accessed and manipulated by code generated for future plan steps.

We have implemented safeguards in the way the CI generates code. We instruct it to generate code within a function with a specific signature. This reduces the likelihood of CI-generated code overwriting global variables containing previous intermediate results during execution. We also provide the CI with the name of the global variable dataframe it should use in its code. We then call the function and attempt to save its results in a different global variable dataframe that would be used in the next plan step.

If the function call encounters an error, the CI receives the error message, its generated code, and the plan step it is trying to solve. It is then prompted to fix the code such that there are no errors. This is done in a loop until there is either no error in the execution or a predefined number of error correction attempts have been reached. In the latter case, the agent returns a message indicating it was unable to generate executable code.

Because the planner module handles most of the heavy lifting, we expect it to generate very explicit plan steps such that the CI does not need to perform any independent reasoning. That is why we designed the CI with minimal system knowledge. We only provide it with the current step it needs to execute (without showing it completed steps or any future steps) and information about the data, such as its contents and column names and descriptions.

4.2.3 Complexity estimation

The problem of automatic tabular data analysis is not trivial. The successful completion of automatic tabular data analysis tasks can be inconsistent. We addressed this by assuming that longer tasks are generally more complex and formulate Hypothesis 1.

To test this hypothesis, we developed several metrics to quantify the complexity of the analysis task using the analysis task query itself, as well as the generated initial and revised plans. All metrics are described in-depth in section 5.2.

The idea is to evaluate complexity (using these complexity metrics) after the initial revision of the plan. This provides more information about the task query, such as an indication of how the planner LLM handles decomposing the task through the plans. Additionally, it gives us a sense of task complexity before querying the CI. This allows us to leverage extra information while limiting calls to our LLMs. Consequently, we can choose to either continue with solving the analysis task or ask the user to simplify it if we deem it too complex for the agent to finish successfully. We refer to this as the gate-keeping mechanism in our agent.

4.2.4 Iterative Feedback Approach

The iterative feedback approach details how the planner and CI work together in a loop, along with the decision-making process for plan revisions. Here is a simplified explanation of the overall process:

Initially the user submits a query, the planner generates a revised plan to solve the query, and this revised plan is then executed using the iterative feedback approach. Pseudo-code 1 illustrates the iterative feedback loop for completing the plan.

The iterative feedback approach is initialized by dividing the revised plan into three parts:

- **Completed Steps**: Stores steps already executed by the CI (empty during the first iteration).
- **Current Step**: Represents the step currently being processed by the CI.
- **Next Steps**: Encompasses all steps following the current step (equal to the revised plan in the first iteration).

After initialization the iterative feedback loop is executed:

1. The first plan step from "Next Steps" is moved to "Current Step."
2. The "Current Step" is passed to the CI for code generation and execution.
3. If errors occur, error correction is attempted.
4. Upon successful execution, intermediate results are obtained (limited to the first ten rows due to context window limitations).
5. The planner receives these intermediate results, along with the original user query, completed plan steps, current step, generated code, and remaining next steps. The planner then revises the remaining next steps if necessary. For example, it might identify the need for additional filtering before proceeding.
6. Regardless of revisions, the "Current Step" is added to "Completed Steps."
7. The loop restarts, processing the next plan step from "Next Steps."

Currently, the CI operates as an isolated module. In some cases, the planning step might be poorly defined (e.g., missing intermediate column names). Even with multiple error correction attempts, the code cannot be fixed. When error correction fails, the agent informs the user it cannot fulfill the request. An improvement could involve a more intertwined error correction process where the planner can provide additional information to the CI to assist in generating working code.

Pseudo-Code 1 Iterative feedback loop

```
1: Initialize:
2:    completed_steps = []
3:    current_step = None
4:    next_steps = initial_revised_plan
5: while next_steps is not empty do
6:     current_step = pop the 1st element of next_steps
7:     error = None
8:     for n_correction do      ▷ Number of times to try error correction in the CI
9:         CI_code = CI.generate(current_step, error)
10:        Execute CI_code
11:        if CI_code execution raised an error then
12:            error = extract error raised in execution
13:        else
14:            break out of for loop on line 8
15:        end if
16:    end for
17:    if error correction failed then
18:        agent_return = "I failed at coding"
19:        break out of while loop on line 5
20:    end if
21:    intermediate_res = first 10 rows of dataframe from CI_code
22:    planner revises next_steps using (
23:      original user query,
24:      completed_steps,
25:      current_step,
26:      generated code,
27:      intermediate_res,
28:      remaining next_steps
29:    )
30:    Append current_step to completed_steps
31: end while
```

We will discuss this improvement opportunity and other potential areas for future work in a later section.

4.2.5 Prompting Strategy

Our system adopts a modular approach, inspired by modular programming software design techniques, to construct prompts for our LLM table analyst agent. This approach involves breaking down our prompts into distinct components, each serving a specific purpose, and utilizing the flexibility of `Python` format strings for dynamic prompt assembly. Such a design enhances the maintainability of our prompt constructing code, as changes to a component only need to be made in one place, thus reducing the likelihood of errors. Furthermore, this approach provides scalability, allowing us to manage complexity as we add, remove, or modify components. One component we reuse is the description of the dataset and its columns. By storing these descriptions separately and inserting them into our prompt template when constructing a prompt, we ensure consistency in how data is referenced across different prompts and allow for centralized updates to the descriptions. When a system prompt or a plan revision prompt is needed, we dynamically assemble it by formatting a base template and inserting the relevant components.

For prompting, we use a base prompt template that outlines the context of our agent's operation, such as focus on tabular data analysis, planner and CI modules. This template is designed with multiple placeholders for various components. There are constant components that are the same for any constructed prompt, such as the data and data column descriptions and the user query. There are also dynamic components that change depending on the task, such as the task description, constraints of the task, and examples for the task. Following our modular approach, these components are dynamically filled in to construct specific prompts for different tasks.

The planner in our agent operates on three distinct tasks: initial planning, initial plan revision, and revision after feedback. While the expected output for each task is the same - a plan represented as a `Python` list of string plan steps; each task has unique goals, constraints, and provided examples. The examples for each task differ because of different inputs.

In the initial planning task, the input is solely the user query, and the planner is tasked with generating a plan that decomposes the user query into executable steps. For the initial plan revision task, the inputs are the user query and the initial plan. The planner's goal is to review the initial plan in the context of the user query, determine its sufficiency, and revise it if necessary. In the revision after feedback task, the inputs are more complex. They include the current step of the plan (the one that was executed), the user query, the generated code, the intermediate result, and other parts of the plan - already completed steps and the next step. The planner is then tasked with determining whether the next steps of the plan require revision based on these inputs.

We also adopt a single prompt-answer interaction approach in our agent, a decision

driven by the desire to simplify our work and reduce the degrees of freedom we need to account for so we can focus on the design of our agent such as the iterative feedback approach described in the previous section. This approach means that our agent does not support a conversational mode of interaction. Instead, it processes each user query independently, without the context of any previous queries. As a result, if a user task lacks crucial information, there is no mechanism to adjust the agent's output based on additional input. The user must provide all necessary details in their initial query. If any crucial information is omitted, the user must rewrite and resubmit their query. The drawback of this approach is the increased importance of comprehensive and precise initial queries, as they directly influence the effectiveness and accuracy of the agent's output.

5

Experiments

All experiments are run on our 50 ablated queries that are described in Section 4.1.1.

5.1 Different Prompting Strategies for the Planner

We investigated the effectiveness of various prompting strategies for the planner module of our LLM agent. Our initial subjective prompt resulted in weaknesses such as the planner separating group-by operations from aggregation operations in different plan steps. Additionally, the aggregation steps themselves were vague and not self-contained, referencing information outside the plan step (e.g., "count tests in resulting dataframe").

To address these shortcomings, we developed a revised prompt that specifically instructed the planner to combine group-by and aggregation operations into single steps. We further specified the use of the `pandas.value_counts()` method for consistency in naming the newly created column. This method returns a `pandas.Series` object named "proportion" if the `normalize` argument is set to `True`, and "count" otherwise. Coercing this `Series` object into a `DataFrame` using methods like `reset_index()` inherits the name of the `Series` for the aggregate column. This approach ensures consistent naming for newly created columns throughout the analysis task.

We further enhanced our revised prompt by incorporating CoT (Wei, Wang, et al., 2022) prompting. This involved adding the line "Let's think step by step." at the end of the prompt, encouraging the LLM to explicitly reason through the steps required to solve the task.

Our observations during experimentation indicated that the initial plan revision process sometimes introduced redundant and ill-defined steps, potentially degrading a good initial plan. To investigate this further, we conducted experiments with the revised CoT prompt and examples, but with plan revision turned off.

We also implemented a self-consistency (SC) (X. Wang et al., 2022) strategy based on prior work, but limitations arose due to the nature of our planner's output. The SC approach in previous research focused on evaluating consistency for arithmetic and commonsense reasoning tasks, where answers are more structured and less variable.

In our case, the planner's output is in natural language containing a plan represented as a `Python` list. This format allows for outputs with the same semantic meaning but phrased differently. This characteristic prevents using a simple majority vote approach on our outputs as employed in the original SC paper.

To address the challenge of evaluating consistency in natural language outputs, we devised an alternative approach. First, the planner generates an initial response A containing descriptive text and a plan. We then prompt the planner to generate a set of n additional responses $(a_i, i \in 1, ..., n)$ with varying generation parameters (`temperature`, `top_p`, `top_k`). We then perform pairwise comparisons between A and each a_i using a separate LLM to determine consistency. Finally, we calculate a consistency confidence score based on the proportion of consistent pairs divided by n:

$$\text{confidence\_score} = \sum_i \frac{\text{is\_consistent}(a_i)}{n}$$

An arbitrary confidence threshold can then be established. Outputs with a SC confidence score exceeding this threshold are considered consistent.

In our implementation, the SC evaluation is run three times with $n = 9$ and a confidence threshold of 0.65. If the planner fails to generate an output with a SC confidence score above this threshold in three attempts, the last generated output is chosen.

We evaluated all three prompting strategies (original prompt, revised prompt, and revised prompt with CoT) with and without examples. The examples consisted of three plans derived from our solutions to easy, medium, and hard tasks (subjectively judged by us). The final prompting strategy, revised prompt with CoT and SC, was only evaluated with examples.

The tasks of the CI were relatively simple compared to the planner's tasks due to the CI not requiring reasoning abilities. This is the primary reason we focused our experimentation solely on varying the planner prompts. Our observations during the experiments revealed that the CI often encountered errors due to ambiguity in the plan step prompts. However, when the plan steps were explicit, the CI successfully generated correct and executable code.

Our agent's workflow can be broadly divided into two stages: planning and execution. In the planning stage, the agent generates and revises an initial plan. The execution stage involves generating, executing, and handling errors in the code needed to complete the plan steps as well as continual plan revision using intermediate results. For most prompting strategies, an average execution time for one query is between 2 to 4 minutes. The planning stage constitutes up to 20 seconds of this run time, showing a skew in execution time towards the execution stage. In contrast, SC shifts the execution time more towards the planning stage due to generation of multiple plans to evaluate consistency. For SC the initial planning stage takes up to 2 minutes and execution of one query can take from 4 to 6 minutes. It is important to note that execution times are reported for an intuitive understanding of the distribution

of computational resources in our agent and is based on our specific hardware setup. It is not meant for absolute performance comparisons.

5.2 Complexity Metrics

We introduce complexity metrics to evaluate Hypothesis (1) regarding the relationship between agent performance and task complexity. These metrics were designed to potentially correlate with task complexity and successful agent completion. We can categorize them into two groups: query-related and plan-related metrics. Query-related metrics solely consider the user query as input, while plan-related metrics additionally leverage the initial and revised plans.

Most metrics, denoted by "(LLM)", involve an LLM for reasoning and information extraction. For these metrics, the LLM receives a more general query to elicit reasoning in its response. For instance, to determine *complex_vocabulary*, the LLM is tasked with identifying technical terms or complex vocabulary within the query. This response is then passed to a separate LLM for extracting the exact count.

5.2.1 Query Metrics

We define the following query-related metrics:

1. Length of query:
   - *query_length* - Character count in query
   - *query_length_words* - Word count in query
2. Query complexity:
   - *query_complex* - Whether the query is complex or not (LLM)
3. Complex or technical vocabulary:
   - *complex_vocabulary* - Number of technical terms or complex words in the query (LLM)
   - *complex_vocabulary_prop* - Proportion of technical terms or complex words in the query
4. Operations:
   - *nested_operations* - Number of nested operations in the query (LLM)
   - *num_operations* - Number of data manipulation operations required to complete the query (LLM)

5.2.2 Plan Metrics

We define the following plan-related metrics:

1. Plan complexity:
   - *plan_complexity* - Number of complex operations in the plans (LLM)
   - *plan_complexity_prop* - Proportion of complex operations in the plans over the length of the longer plan
2. Plan length:
   - *init_plan_length* - Length of the initial plan

- *rev_plan_length* - Length of the revised plan
- *rev_plan_length_increase* - $\frac{\text{rev\_plan\_length}}{\text{init\_plan\_length}}$

3. Initial and revised plan similarity
   - *sequence_similarity* - Flattening plan lists of subqueries into lists of words and running `Python`'s `difflib.SequenceMatcher()` on the 2 word lists
   - *sequence_similarity_llm_num* - Asking the LLM for a similarity score of "low", "medium" or "high" for the two plans, which is represented as 1, 2 and 3 respectively (LLM)
   - Plan embedding similarity (these are explained in the following paragraphs, each use a different approach to calculating similarity):
     (a) *embedding_similarity1_mean, embedding_similarity1_median* - Approach 1
     (b) *embedding_similarity2_mean, embedding_similarity2_median* - Approach 2
     (c) *embedding_similarity3* - Approach 3
     (d) *embedding_similarity4* - Approach 4

5.2.3 Plan Embeddings

Table 5.1: Approaches to embedding plans

Approach 1 1. Encode each step of both plans. 2. Adjust for differing length of plans by padding the shorter plan with zero vectors. 3. Calculate pair-wise distances for each pair of plan steps. 4. Aggregate (mean or median) pair-wise distances and convert to similarity.	**Approach 2** 1. Encode each step of both plans. 2. Aggregate (mean or median) the embeddings for each plan. 3. Compute similarity.
Approach 3 1. Encode each step of both plans. 2. Use Dynamic Time Warping (DTW) to calculate distance and convert it to similarity. It can handle different length sequences, i.e., plans with different number of steps.	**Approach 4** 1. Concatenate all steps of a plan into one long string. 2. Encode the long string plans. 3. Compute similarity of embeddings.

The plan embedding similarity metrics utilize an encoder model to transform both

plans into numerical representations (embeddings). Subsequently, the distance between these embeddings is compared to assess plan similarity. The underlying hypothesis and motivation for these metrics is that the planner can generate optimal plans for simple tasks on the first attempt (initial plan). Conversely, for more complex tasks, the planner might require revision to improve the initial plan. Consequently, similar plans would suggest simpler tasks, whereas dissimilar plans might indicate more complex tasks.

Four unique approaches to plan embedding were explored, as detailed in Table 5.1. Both cosine distance and cosine similarity were employed as distance and similarity metrics.

5.3 Dataframe Similarity

Our agent outputs dataframes as a result of its iterative approach to solving tasks. This approach involves decomposing tasks into smaller steps represented by a plan. To ensure consistency of intermediate results after each plan step execution, we use dataframes as the storage structure for both intermediate and final results. This choice is further reinforced through our prompting strategy.

Since we also solve all tasks in our dataset manually, generating ground-truth code and resulting dataframes, evaluating our agent's performance requires a measure of similarity between the agent-generated output dataframe and the corresponding ground-truth dataframe.

We have developed a function that calculates a dataframe similarity score between two dataframes, ranging from 0 to 1. Higher scores indicate greater similarity. The function also incorporates error handling for specific failure modes that prevent similarity calculation:

- **-1**: No common columns exist between the two dataframes.
- **-5**: Input arguments are not dataframe or series objects.
- **-9**: The agent crashed and did not produce a final result.

The dataframe similarity function is described in pseudo-code 2 and takes two dataframes, `df1` and `df2`, as input. It first verifies if the inputs are dataframes or series objects. Series objects are converted to dataframes, while other data types return failure mode -5. The function then removes duplicate columns, keeping only the rightmost column. This is done because the agent sometimes renames an existing column to match the name of a newly created column (e.g., "count" from using `value_counts()`). Next, the function identifies common columns between the dataframes. If no common columns are found, failure mode -1 is returned.

It is important to note that our dataframe similarity calculation considers only columns present in both dataframes. This means that if one dataframe has one column and the other has 100 columns, but the single common column perfectly matches, our function would return a similarity score of 1. However, in our context, this behavior is acceptable. Verification engineers prioritize the presence of the correct answer over the absence of irrelevant data. Additionally, selecting all columns

Pseudo-Code 2 Dataframe Similarity Function

```
Function dataframe_similarity(df1, df2):
    If df1 or df2 is not a DataFrame:
        If df1 or df2 is a Series:
            Convert to DataFrame
        Else:
            Return -5.0
    Drop duplicate columns in df1 and df2
    common_columns = Find common columns between df1 and df2
    If no common columns:
        Return -1.0
    Select common columns in df1 and df2 and drop duplicate rows
    Pad df1 and df2 with None to equal length
    Initialize label encoder (LE)
    Initialize similarity_score
    For each column in common_columns:
        If either column is empty, continue
        Convert column values to string and replace NaN values with "NA"
        Fit label encoder and transform column values to numeric
        similarity_score += Jaccard_score(df1[column], df2[column])
    similarity_score /= length(common_columns)
    Return similarity score
```

in our case is not a problem because we initially only select core relevant columns for analysis. Having all of the core columns in the output does not pose a problem according to our domain expert. It is worth noting that this approach could be an issue in other applications where the agent might return a large number of potentially irrelevant columns.

After dropping duplicate rows, the function addresses potential differences in dataframe lengths by padding both dataframes with `None` values to achieve equal length for comparison. This is achieved through a full outer join on all values in the common columns of both dataframes. This approximates finding the common rows between the two dataframes. Dataframe indices are not used for this purpose because they can be dropped during agent operations, leading to unreliable results. After the full outer join, rows present only in one dataframe are replaced with `None` values in the other dataframe (e.g., rows in `df1` only present in `df2` are replaced with `None` in `df1`). This step ensures both dataframes have the same length and row order.

To ensure robustness for various data types and missing values, the function utilizes a label encoder (LE). The LE iterates over each common column (excluding empty columns), converts column values to strings, and replaces `NaN` values with "NA". These strings are then transformed into numerical representations using the LE. This approach effectively handles both numerical and categorical columns.

Our dataframe similarity metric leverages the Jaccard score. Traditionally used

to measure similarity between sets, the Jaccard score is defined as the size of the intersection divided by the size of the union of the two sets. In the context of machine learning classification, the Jaccard score is applied to lists of labels, where it represents the proportion of true positives to the sum of true positives, false positives, and false negatives. This application differs slightly from its original set-based usage as it considers element order and allows for duplicates, common in label data.

The Jaccard score is computed using the `scikit-learn` library for each pair of common columns between the two dataframes. Since this implementation is designed for classification model evaluation, it treats all values in a column as class labels. As the Jaccard score is both symmetric and supports multi-class classification, it is suitable for our scenario with columns containing diverse values.

After iterating over all common column pairs, the total Jaccard score is divided by the number of common columns to obtain the average Jaccard score. This average score is then returned as the final dataframe similarity score.

This function provides a robust method for calculating the similarity between two dataframes. It accounts for potential variations in column names, data types, and dataframe lengths. Additionally, it incorporates error handling for non-dataframe inputs and provides informative failure mode values for specific error scenarios.

5.4 Task solution complexity

Evaluating the inherent complexity of an analysis task solely based on its natural language description is a well-recognized challenge as described in Section 3.6. To address this, we opted to calculate the complexity of our ground-truth code solutions. We adapted the approach presented in (Wood, 1986) to our specific context, focusing on component complexity and coordinative complexity. Dynamic complexity was excluded as our system operates in a static manner, meaning the component and coordinative complexity of the generated code remains constant over time. In simpler terms, our system does not revisit and regenerate previously generated and functional code.

Component complexity refers to the acts and information cues involved in solving the task. In our case, these translate to functions (including class methods) and data columns, respectively. We calculate the component complexity by assigning a value of 1 for each distinct function encountered and an additional 1 for each unique data column referenced within that function. Importantly, if the same column appears in multiple functions, it is counted towards the complexity of each function individually. For instance, filtering a column (`column1`) on specific values (`val1` and `val2`) would result in a component complexity of 2. This accounts for the single distinct function (filter) and the single distinct column (`column1`). Similarly, calculating the frequency of values in `column1` using a function like `value_counts()` would also yield a component complexity of 2 (one function and one column), even though `column1` is included in the complexity calculation for both functions.

Coordinative complexity captures the dependency between the identified acts (functions). In our context, this translates to the dependency or sequential nature of function execution. We assign a value of 1 to the coordinative complexity for each function that relies on the output of the preceding function. For example, data must be filtered before calculating value frequencies. Due to the inherent linear execution flow of our agent, each function inherently depends on the previous one. Consequently, in our system, the coordinative complexity simplifies to the total number of distinct functions minus 1, reflecting the absence of branching logic in our execution pipeline.

5.5 Domain expert evaluation

To gain further insights into our agent's performance, we asked for feedback from a domain expert - a verification engineer. The expert was provided with the following information for each task:

- **Analysis Task (Query)**: The natural language description of the analytical task.
- **Agent Code (Python)**: The code generated by the agent to solve the task.
- **Agent Output (CSV)**: The dataframe generated by the agent exported as a CSV file for ease of analysis.

The expert was instructed to evaluate the agent's output dataframe in the context of the given analysis task using a color-based rating system:

- **Green**: The output precisely matches the expert's expectations for the task.
- **Yellow**: The output is mostly correct, but may contain minor omissions or inconsistencies.
- **Red**: The output is significantly flawed and unsuitable for the task.

To assess the validity of our dataframe similarity metric, we performed a correlation analysis between its output and the domain expert's evaluations. To facilitate this analysis, the color ratings ("green," "yellow," and "red") were converted into numerical scores (3, 2, and 1, respectively), with higher values indicating greater similarity to the desired outcome.

Furthermore, we described the dataframe similarity function to the domain expert and sought their professional opinion on its effectiveness in measuring similarity between dataframes in the context of our task domain. This feedback will provide valuable insights into the metric's strengths and potential areas for improvement.

6

Results and Discussion

6.1 Dataframe Similarity: A Domain-Specific Evaluation Metric

Evaluating the performance of an LLM agent designed for a specific domain like software test data analysis in the automotive industry can be challenging with generic benchmarks. These benchmarks often focus on broad language understanding tasks and might not capture the intricacies of real-world analysis processes within your domain. Here, we introduce the Dataframe Similarity metric, a novel approach that addresses these limitations.

This section explores the effectiveness of various prompting strategies in achieving both agent completion and high quality output dataframes, as measured using our Dataframe Similarity metric. Furthermore, we validate our Dataframe Similarity metric in collaboration with a domain expert.

Table 6.1: Dataframe similarity and agent completion for all prompting strategies

	DF Similarity		Completion	
	Replaced	Filtered	N	Rate
Original	0.10	0.51	11	0.22
Original + examples	0.19	0.68	14	0.28
Revised	0.17	0.54	21	0.42
Revised + examples	0.28	0.48	29	0.58
Revised CoT	0.14	0.63	14	0.28
Revised CoT + examples	0.38	0.76	25	0.50
Revised CoT + examples + no revision	**0.56**	**0.85**	**34**	**0.68**
Revised CoT SC + examples	0.32	0.80	21	0.42
Revised CoT SC + examples + no revision	**0.56**	**0.85**	**34**	**0.68**

The results in Table 6.1 highlight the effectiveness of examples in improving all prompting strategies. Regardless of the prompting strategy employed, including examples consistently led to better performance in terms of both dataframe similarity and agent completion. This is why we opted against running a zero-shot experiment for the SC strategy ("Revised CoT SC ...").

Interestingly, our plan revision step did not yield the anticipated improvements.

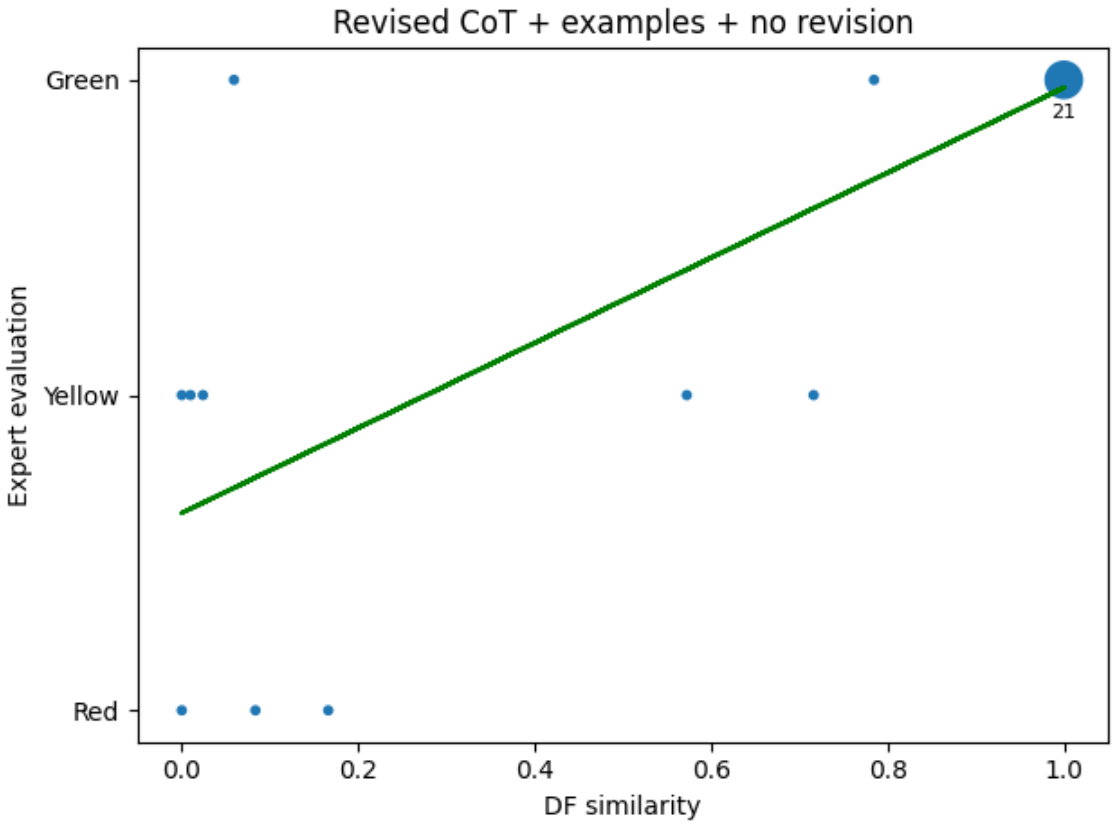


Figure 6.1: Expert evaluation vs Dataframe similarity score

In fact, the best performing strategy excluded this revision process entirely. This strategy, "Revised CoT + examples + no revision", achieved the highest scores across all metrics. This finding contrasts with the common agent architecture identified in **RQ1**, where iterative plan revision based on intermediate results is a prevalent approach. While plan revision offers theoretical benefits, in the context of our LLM agent it seems to even hinder performance.

Our implementation of SC did not lead to any noticeable improvements in the CoT strategy. We consistently observed a SC confidence score of 1 for all queries, indicating that the planner always generated consistent initial plans. This suggests that our reworded queries were unambiguous enough for the planner to produce consistent plans without revision. Consequently, "Revised CoT SC + examples + no revision" became identical to our best performing strategy, implying that SC did not improve our plans in this context.

6.1.1 Domain Expert Evaluation

We selected the best performing prompting strategy, "Revised CoT + examples + no revision", for which our agent produced final outputs for 34 out of 50 ablated queries. We provided the queries, generated code, and generated output from this strategy to a domain expert. The expert discarded 3 ablations deemed nonsensical in real-world applications, evaluating the remaining 31. His evaluations were coded

as "red," "yellow," and "green," corresponding to 1, 2, and 3, respectively. Pearson correlation analysis between the expert evaluation and dataframe similarity scores revealed a coefficient of 0.82 with a p-value of 1.98×10^{-8}.

These results are visualized in Figure 6.1 along with a linear regression line. The majority of the data points (21 out of 31) were rated as "green" and have a perfect dataframe similarity score of 1.0. The remaining 10 points are scattered throughout the plot. This significant skew towards the top right corner could potentially influence the high correlation coefficient.

Additionally, we described our dataframe similarity function to the domain expert and inquired about its fairness in evaluating agent results. The expert agreed that verification engineers prioritize filtering the correct rows, and extra irrelevant columns are acceptable as long as all necessary rows are present. This aligns with the high correlation coefficient, suggesting that our dataframe similarity score captures aspects of agent performance that are also important to domain experts.

6.1.2 Answering Research Question 2

By introducing the Dataframe Similarity metric and demonstrating its effectiveness through expert validation, this section directly addresses Research Question 2:

> **RQ 2:** How can we evaluate the performance of a domain-specific application of an LLM agent designed for analyzing software test data, considering the limitations of generic evaluation benchmarks?

The Dataframe Similarity metric, along with the expert evaluation, provides a valuable tool for assessing the performance of our LLM agent in the context of software test data analysis tasks. While the current experiments did not explore additional metrics or delve deeper into the limitations of generic benchmarks for this specific domain, the presented findings offer a strong foundation for future research in this direction.

6.2 Task Ablations

Table 6.2: Dataframe similarity and agent completion for task ablations

	DF Similarity	Completion			
Up to Subquery	Mean	Success	Failure	Total	Rate
1	0.88	15	1	16	0.94
2	0.95	11	5	16	0.69
3	1.00	4	8	12	0.36
4	0.09	3	3	6	0.50

We now analyze the performance of our agent on task ablations (subqueries) using the best performing prompting strategy, "Revised CoT + examples + no revision".

Table 6.2 presents the results, including dataframe similarity and agent completion for each subquery.

While the results in Table 6.2 do seem to support Hypothesis 1, which predicted a higher completion rate for earlier subqueries due to lower complexity, it is important to consider the limitations of our study. The sample size of only 50 queries is too small to draw statistically significant conclusions.

Additionally, we looked deeper into the results of our best performing strategy ("Revised CoT + examples + no revision") on our original 15 analysis tasks (use-cases). The agent successfully completed 8 tasks, achieving varying degrees of success based on our dataframe similarity metric. Four queries had a dataframe similarity score of 1, the other four had scores of 0.5, 0.19, 0.05 and 0.03. This translates to a success rate of 26.7% considering only queries with dataframe similarity score of 1 or 53.3% overall. This demonstrates the agent's potential as a productivity tool in data analysis in the automotive industry.

6.3 Complexity Metrics

Table 6.3: Correlation with agent failure

	correlation	p-value
embedding_similarity3	-0.530	0.000
rev_plan_length	0.465	0.001
embedding_similarity1_mean	-0.457	0.001
sequence_similarity	-0.393	0.005
embedding_similarity2_median	-0.361	0.010
embedding_similarity2_mean	-0.335	0.017
rev_plan_length_increase	0.294	0.038
plan_complexity	0.283	0.049
embedding_similarity4	-0.262	0.066
embedding_similarity1_median	-0.251	0.079
init_plan_length	0.200	0.164
complex_vocabulary_prop	0.183	0.204
plan_complexity_prop	-0.107	0.463
complex_vocabulary	0.081	0.578
sequence_similarity_llm_num	-0.071	0.626
nested_operations	-0.054	0.715
query_complex	0.040	0.783
query_length_words	-0.022	0.878
query_length	0.010	0.943
num_operations	-0.000	1.000

Our complexity metrics, described in Section 5.2, were calculated for the "Revised prompt CoT + examples" prompting strategy due to their reliance on differences between initial and revised plans. This choice ensured meaningful results, as metrics

Table 6.4: Correlation with dataframe similarity (failure modes replaced with 0)

	correlation	p-value
embedding_similarity3	0.478	0.000
rev_plan_length	-0.397	0.004
embedding_similarity1_mean	0.394	0.005
embedding_similarity2_median	0.371	0.008
sequence_similarity	0.371	0.008
plan_complexity	-0.308	0.031
embedding_similarity2_mean	0.284	0.045
rev_plan_length_increase	-0.247	0.084
complex_vocabulary_prop	-0.209	0.146
embedding_similarity1_median	0.194	0.178
embedding_similarity4	0.190	0.187
init_plan_length	-0.123	0.393
complex_vocabulary	-0.091	0.528
sequence_similarity_llm_num	0.084	0.560
num_operations	-0.067	0.649
query_complex	0.046	0.749
query_length_words	0.045	0.758
plan_complexity_prop	-0.036	0.804
query_length	0.014	0.924
nested_operations	0.003	0.986

for strategies without plan revision would not be applicable. We now present the results of Pearson correlation analysis between these metrics and various targets for analysis, including:

- Agent failure rate (Table 6.3)
- Dataframe similarity with failure modes replaced by 0 (Table 6.4)
- Dataframe similarity with failure modes filtered out (Table 6.5)
- Component complexity (Table 6.6)
- Coordinative complexity (Table 6.7)
- Total unweighted complexity (Table 6.8)

All tables are sorted by the absolute value of the correlation coefficient, and p-values are color-coded for significance: green (p-value <= 0.05), orange (0.05 < p-value <= 0.1), and red otherwise.

6.3.1 Agent Failure Target

Table 6.3 reveals a positive correlation between the revised plan length, its increase compared to the initial plan, and agent failure. Assuming more complex tasks are decomposed into longer plans the necessitate further revision, this aligns with Hypothesis 1. However, these results depend on the specific planner not the task itself, and a different design of the planner might produce different correlations.

Table 6.5: Correlation with dataframe similarity (failure modes filtered out)

	correlation	p-value
plan_complexity_prop	-0.248	0.232
plan_complexity	-0.225	0.279
embedding_similarity2_median	0.204	0.328
query_complex	0.195	0.349
num_operations	-0.173	0.420
complex_vocabulary_prop	-0.153	0.464
embedding_similarity3	0.153	0.467
sequence_similarity	0.122	0.561
embedding_similarity1_median	-0.111	0.599
init_plan_length	0.099	0.636
nested_operations	-0.093	0.660
embedding_similarity1_mean	0.069	0.745
embedding_similarity4	-0.067	0.752
complex_vocabulary	-0.061	0.774
sequence_similarity_llm_num	0.058	0.783
query_length_words	0.056	0.789
query_length	0.050	0.811
rev_plan_length	-0.046	0.827
embedding_similarity2_mean	0.029	0.891
rev_plan_length_increase	-0.018	0.930

Most embedded plan similarity measures exhibit statistically significant negative correlations, suggesting that substantial initial plan revision increases the likelihood of agent failure. This could relate to inherent task complexity captured by the initial plan that does not require revising, or potential revision failures that sabotage the initial plan.

6.3.2 Dataframe Similarity Target

The most prominent correlations in Table 6.4 are again related to plans and their similarity. All statistically significant complexity metrics from the agent failure analysis are also present here. This makes sense because agent failure is reflected in the dataframe similarity score as 0.

Furthermore, analyzing only successful completions in Table 6.5 (where failure modes have been filtered out) reveals no significant correlations with complexity metrics. This suggests that these metrics are not suitable for predicting the quality of the final output, but they might be useful for indicating whether the agent can generate an output at all.

Table 6.6: Correlation with component complexity

	correlation	p-value
num_operations	0.719	0.000
query_length_words	0.715	0.000
query_length	0.703	0.000
init_plan_length	0.693	0.000
query_complex	0.686	0.000
complex_vocabulary	0.633	0.000
complex_vocabulary_prop	0.464	0.001
rev_plan_length_increase	-0.393	0.005
embedding_similarity2_mean	0.364	0.009
embedding_similarity4	0.310	0.028
rev_plan_length	0.303	0.032
embedding_similarity2_median	0.282	0.047
nested_operations	0.234	0.110
sequence_similarity	0.231	0.107
plan_complexity	0.201	0.166
plan_complexity_prop	-0.176	0.225
embedding_similarity3	0.134	0.355
sequence_similarity_llm_num	0.090	0.533
embedding_similarity1_median	-0.085	0.559
embedding_similarity1_mean	0.004	0.975

6.3.3 Task Complexity Target

Tables 6.6, 6.7, and 6.8 present correlations with task solution complexity (component, coordinative, and total unweighted, respectively). These correlations are remarkably similar in magnitude and significance, likely due to our method of decomposing solutions into simple steps, the linear nature of our task execution and the way we defined both component and coordinative complexity.

The high correlation with query length (characters or words) suggests that longer queries allow for specifying more operations and columns. Similarly, we observe high correlation with the number of recognized complex vocabulary terms (count and proportion) or the number of operations in the query. This could exemplify the LLM's ability to recognize data manipulation operations and words that refer to column names in a query, which is something we capture with our calculated complexity metrics. The correlation with the binary variable *query_complex* indicates a potential alignment between the LLMs perception of task complexity and the number of operations (functions, methods), information cues (columns), and sequential dependencies. Given that LLMs have been trained on large text corpora, this observation could provide support for the original task complexity research by linking it with the concept of complexity inherent in the training corpora.

The length of both plans correlates with complexity, as each step introduces additional components and sequential dependencies that contribute to our component

Table 6.7: Correlation with coordinative complexity

	correlation	p-value
init_plan_length	0.705	0.000
query_complex	0.681	0.000
query_length_words	0.634	0.000
num_operations	0.618	0.000
query_length	0.591	0.000
complex_vocabulary	0.550	0.000
rev_plan_length_increase	-0.476	0.000
complex_vocabulary_prop	0.389	0.005
embedding_similarity2_mean	0.359	0.010
embedding_similarity4	0.283	0.046
embedding_similarity2_median	0.276	0.052
nested_operations	0.274	0.059
sequence_similarity	0.251	0.079
rev_plan_length	0.226	0.114
plan_complexity_prop	-0.189	0.192
embedding_similarity3	0.162	0.260
embedding_similarity1_median	-0.133	0.358
embedding_similarity1_mean	-0.096	0.508
plan_complexity	0.081	0.578
sequence_similarity_llm_num	0.027	0.851

and coordinative complexity measures. Interestingly, the initial plan length has a stronger correlation with complexity than the revised plan length. This suggests that the initial plan better aligns with our calculated complexity rather than the revised plan.

The correlation observed with plan embedding similarity suggests a possible association between the lack of revision in the initial plan and higher task complexity. This could reflect the LLM's capability to extract the necessary operations and columns from the analysis task. Finally, the negative correlation of the revised plan length increase suggests that the agent might get stressed and introduce unnecessary steps due to a bias towards making changes. Alternatively, for both revised plan length increase and embedding similarity, the results could suggest that tasks of lower complexity are easier to optimize, leading to more frequent revisions.

6.3.4 Overall Discussion of Metrics

Query length metrics were included for completeness, although no correlation with agent failure or dataframe similarity was expected due to their simplicity. However they did show significant correlation with our calculations of task complexity. This is unsurprising, as longer queries have more space to specify additional operations and columns, which directly affect our calculated complexity.

Table 6.8: Correlation with total complexity

	correlation	p-value
init_plan_length	0.706	0.000
query_length_words	0.699	0.000
num_operations	0.695	0.000
query_complex	0.694	0.000
query_length	0.676	0.000
complex_vocabulary	0.615	0.000
complex_vocabulary_prop	0.446	0.001
rev_plan_length_increase	-0.424	0.002
embedding_similarity2_mean	0.367	0.009
embedding_similarity4	0.306	0.031
embedding_similarity2_median	0.284	0.045
rev_plan_length	0.283	0.046
nested_operations	0.250	0.087
sequence_similarity	0.240	0.093
plan_complexity_prop	-0.183	0.208
plan_complexity	0.166	0.255
embedding_similarity3	0.144	0.317
embedding_similarity1_median	-0.101	0.485
sequence_similarity_llm_num	0.071	0.622
embedding_similarity1_mean	-0.027	0.851

Other metrics focusing on the LLM's identification of complexity within the query, such as *query_complex*, *complex_vocabulary*, *nested_operations*, and *num_operations*, could potentially provide insights into the LLM's ability to discover and extract complexities. One would naturally expect that being able to identify these complexities would also allow the LLM to decompose them into a valid plan, leading to lower agent failure rates and higher quality output dataframes.

While there were significant correlations for metrics related to the LLM identifying complexity with our calculation of task complexity, there were no significant correlations with agent failure or dataframe similarity. This suggests that while an LLM is capable of identifying complexity in the query, it might not be as adept at decomposing it into actionable steps.

We derived an assumption from Hypothesis 1, that tasks decomposable into longer plans are more complex. This motivated calculating the lengths of the plans generated by our planner. While the initial plan length had a stronger correlation with task complexity than the revised plan length, the opposite was true for agent failure and dataframe similarity - the revised plan length had a stronger correlation. The task complexity correlation with the initial plan and agent failure correlation with the revised plan might suggest that the initial plan was more aligned with the actual task and revising the plan led to more failures.

Finally, we investigated various plan similarity metrics, including plan embedding

similarity, motivated by the assumption that plan revision might indicate more inherent complexity in the analysis task that the initial decomposition could not capture. However, supporting results do not necessarily confirm our assumption; they only suggest a relationship between our planner choosing to revise the initial plan and the agent system failing. For example, while plan embedding similarity is correlated with agent failure, this does not necessarily mean that the task is more complex. It suggests a potential relationship between the revision of the initial plan and the agent failing more often, reflecting the LLM's perception of the task rather than the task complexity itself.

6.3.5 Answering Research Question 3

By introducing complexity metrics derived from the LLM's understanding of various forms of complexity in the user query and the characteristics of the plans it generates, and performing correlation analysis, this section directly addresses Research Question 3:

> **RQ 3:** How can analysis task complexity be measured from a natural language query, and how does it influence the performance of a domain-specific LLM agent for tabular data analysis?

In conclusion, most metrics derived from LLMs are proxies that capture how the LLM perceives the task. It is challenging to directly link agent performance with task complexity because capturing the difficulty of an analysis task query solely from its natural language representation is inherently difficult. Even human data analysts can attest to underestimating the complexity of an analysis task at first glance. The complexities often only become apparent when they start executing the task.

6.4 Ethics discussion

6.4.1 Power consumption

LLMs are by definition *large*, having billions and even trillions of parameters. Developing and using them requires vast amounts of energy. Patterson et al. (2021) calculated the training of GPT3 to consume 1287 MWh of electricity and produce 552 tonnes of CO_2. It is also stated that training only makes up about 10-20% of the machine learning workload at companies such as Amazon Web Services and Nvidia, meaning that the life-time energy consumption of a model can be orders of magnitudes larger. For reference, according to the United States Environmental Protection Agency[1], a typical passenger vehicle emits about 4.6 tonnes of CO_2 per year. That means that only the training of GPT3 (which most likely is orders of magnitude smaller than GPT4)[2] emits as much CO_2 as 120 typical cars over a

[1] https://www.epa.gov/greenvehicles/greenhouse-gas-emissions-typical-passenger-vehicle

[2] The technical details regarding architecture, model-size, and training of GPT4 has not been made public by OpenAI, citing the "competetive landscape" and "safety implications" in their report (OpenAI, 2023a, p. 2). Even so, the statement is not controversial. The report devotes an

year of typical driving. Estimating inference costs for ChatGPT, Ludvigsen (2023) arrives at 4,167 MWh per month, and that was before the launch of GPT4.

This is of course a major concern and there are numerous approaches to making the models, algorithms, and infrastructure more energy efficient. *Quantization* of the parameters means that you reduce the number of bits of information in each parameter value, normally from the original 32 bits to only eigth or even four bits. This has been shown to massively decrease computation cost while maintaining most of the performance, depending on the model and use case (Zhu et al., 2023).

The algorithm for attention in early Transformers suffers from quadratic time complexity. The original paper focused on translating short paragraphs but with modern context windows of over 100,000 tokens such complexity is problematic. There are a number of approaches to address this problem, for more details on the complexity of self-attention and an overview of alternatives the reader is referred to Keles et al. (2023).

In parallel with making the models and algorithms more computationally efficient, the infrastructure for training and inference are becoming more efficient (Patterson et al., 2021) and are in quite a high degree powered by renewable energy sources. Both Microsoft and Google have the stated goal, at the time of writing, to be entirely powered by carbon free energy sources by 2030. Google has actually made quite some progress and had over 50% of datacenter electricity from carbon free sources already in 2019 (Strubell et al., 2019) and report they reached 65% in 2022 [3].

6.4.2 Automation of work

Advances in technology and machinery has profoundly changed the landscape of labor and productivity over centuries, reducing the number of workers required for certain tasks by many orders of magnitude. Historically it is the *routine* tasks that have been automated, tasks that can be completed by following a series of explicit rules or movements. Technological advances over the last couple of decades has however enabled automation of numerous *non-routine* tasks, tasks that can be both manually and cognitively intensive. Frey and Osborne (2017) provide an account of the history of technological revolutions and employment, and lists recent examples of now automated non-routine tasks that only recently before had been thought to be near-impossible to automate. The examples include, among other things, cars that drive themselves surprisingly well, bank fraud detection algorithms, cancer treatment diagnostics analyses, and robots that handle logistics inside hospitals.

Introducing a task-based framework for discourse on automation and its impact on tasks, Acemoglu and Restrepo (2018) propose that although advancements in AI technology can produce a powerful *displacement* effect in the labor market, the

entire section to "Predictable scaling", in which their methods for predicting GPT-4 performance from models trained with $1,000 \times -10,000\times$ less compute is presented. GPT-4 is also presented as "the latest milestone in OpenAI's effort in scaling up deep learning" on the GPT-4 product research page (OpenAI, 2023b).

[3] https://sustainability.google/progress/energy/

resulting *productivity* effects generally countervail such displacement by increasing labor demand for non-automated tasks and by creating completely new tasks.

This thesis addresses automation of a subset of possible analysis tasks, subjectively validated by a domain expert as a tool for saving time, and as such will not directly replace any human employees. Its potential lies in a potential productivity increase for verification engineers.

6.5 Alternative solutions

An LLM-agent is not the only way to approach the automation problem presented in this thesis. The research fields of *natural language interface (NLI) to database* (Kim et al., 2020) and *NLI for visualization* (Shen et al., 2022) have explored the problem with different end-goals in mind and with focus on different sub-problems. NLI-research has leveraged the capacity of LLMs by having the LLM generate SQL-code directly from the question (Dong et al., 2023; Li et al., 2024) as well as after decomposing the question into sub-problems (Pourreza & Rafiei, 2023). To apply NLI-methods on the Gonogo datasheet one would have to adapt either the data to fit the research, or the methods to apply to Excel and Python rather than SQL.

7

Conclusion

In this thesis, we presented a novel table analyst LLM agent with a two-module architecture: a planner module and a CI module. The planner generates a step-by-step plan for analyzing a table, and the CI module executes each step individually. Additionally, we developed metrics to predict the complexity of the analysis task and evaluate the agent's output against ground truth.

7.1 Key Findings

Our research identified that including examples within prompting strategies consistently improved the agent's performance. Interestingly, plan revision did not yield any significant benefits, with the best strategy actually excluding it entirely ("Revised CoT + examples + no revision"). Similarly, implementing Self-Consistency offered no improvements, suggesting the planner generated consistent initial plans. Importantly, a domain expert validated our dataframe similarity metric, which was calculated for the "Revised CoT + examples + no revision" strategy. This metric also achieved a high correlation (0.82) with expert ratings, indicating its effectiveness in capturing relevant aspects of quality within the agent's outputs.

While the results on task ablations using the "Revised CoT + examples + no revision" strategy seem to support Hypothesis 1 of higher completion rates for earlier subqueries (due to assumption of lower complexity), the sample size (50 queries) is too small for statistically significant conclusions.

Analysis of complexity metrics suggest that LLMs may be able to recognize complexity in an analysis query but may struggle to decompose and translate the query into actionable plan steps. Moreover, the analysis suggests these complexity metrics capture the LLM's perception of task complexity rather than inherent task complexity. Capturing intrinsic task complexity from natural language queries remains a challenge.

Additionally, analysis of plan focused complexity metrics suggest that initial plan revision leads to worse agent performance. This could be due to a failure of the revision process or suggest that complex queries are more difficult to revise a plan for while simpler queries are easier to optimize in the plan revision.

7.2 Future Work

Future research can explore several avenues to improve the agent's understanding of task complexity. This could involve exploring different complexity metrics or investigating the link between intrinsic task complexity and the LLM's perception of task complexity, which could lead to better metrics that accurately reflect the true challenge of a task.

Beyond the current approach of generating and revising a single plan, future work could investigate alternative plan generation strategies. One approach involves generating and selecting from multiple initial plans based on predicted success rates or some measure of plan quality. This could lead to more robust plan selection compared to the current single-plan generation and revision approach. While another beneficial approach could be a ReAct-inspired iterative plan generation design. Here, the planner would generate only the first step of the plan (Reason), execute it (Act), observe the intermediate result (Observe), and then use this information to determine subsequent steps. This approach might be particularly suitable for complex tasks where a complete decomposition of the problem is challenging and early feedback can significantly improve plan direction.

Furthermore, future work could explore additional avenues to enhance the agent's capabilities. Integrating the planner into the CI's error correction mechanism could improve the agent's robustness in handling unexpected errors. Additionally, implementing a chat function that allows users to interact with the agent conversationally would enable real-time refinement of results through clarification requests and feedback. This could be particularly valuable for complex tasks where user interaction can guide the agent towards a successful solution.

7.3 Conclusion

This work demonstrates the potential of LLM agents as productivity tools for partially automating ad-hoc table analysis tasks in the automotive industry. While full automation remains a future goal, the development of complexity prediction and evaluation metrics provides a valuable framework for further research and improvement of LLM agents. These metrics can guide the development of more robust and efficient agents that can help to improve analysts' efficiency by automating a portion of the analysis process.

Bibliography

Abu-Akel, A. M., Apperly, I. A., Wood, S. J., & Hansen, P. C. (2020). Re-imaging the intentional stance. *Proceedings of the Royal Society B*, *287*(1925), 20200244.

Acemoglu, D., & Restrepo, P. (2018). Artificial intelligence, automation, and work. In *The economics of artificial intelligence: An agenda* (pp. 197–236). University of Chicago Press.

Almaatouq, A., Alsobay, M., Yin, M., & Watts, D. J. (2021). Task complexity moderates group synergy. *Proceedings of the National Academy of Sciences*, *118*(36), e2101062118. https://doi.org/10.1073/pnas.2101062118

Androutsopoulos, I., Ritchie, G. D., & Thanisch, P. (1995). Natural language interfaces to databasesan introduction. *Natural language engineering*, *1*(1), 29–81.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. (2021). Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, Y., Ducharme, R., & Vincent, P. (2000). A Neural Probabilistic Language Model. In T. Leen, T. Dietterich, & V. Tresp (Eds.), *Advances in neural information processing systems* (Vol. 13). MIT Press. https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., ... Amodei, D. (2020). Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 1877–1901, Vol. 33). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

Buccino, G., Lui, F., Canessa, N., Patteri, I., Lagravinese, G., Benuzzi, F., Porro, C. A., & Rizzolatti, G. (2004). Neural circuits involved in the recognition of actions performed by nonconspecifics: An fMRI study. *Journal of cognitive neuroscience*, *16*(1), 114–126.

Chang, Y., Wang, X., Wang, J., Wu, Y., Zhu, K., Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., et al. (2023). A Survey on Evaluation of Large Language Models. *arXiv preprint arXiv:2307.03109*.

Chen, X., Lin, M., Schärli, N., & Zhou, D. (2023). Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Cheng, Y., Zhang, C., Zhang, Z., Meng, X., Hong, S., Li, W., Wang, Z., Wang, Z., Yin, F., Zhao, J., et al. (2024). Exploring Large Language Model based Intelligent Agents: Definitions, Methods, and Prospects. *arXiv preprint arXiv: 2401.03428*.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J. E., et al. (2024). Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., ... Fiedel, N. (2023). PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research*, *24*(240), 1–113. http://jmlr.org/papers/v24/22-1144.html

Danner-Schröder, A., & Ostermann, S. M. (2022). Towards a Processual Understanding of Task Complexity: Constructing task complexity in practice. *Organization Studies*, *43*(3), 437–463. https://doi.org/10.1177/0170840620941314

Dasgupta, I., Kaeser-Chen, C., Marino, K., Ahuja, A., Babayan, S., Hill, F., & Fergus, R. (2023). Collaborating with language models for embodied reasoning. *arXiv preprint arXiv:2302.00763*.

Dennett, D. C. (1989). *The intentional stance*. MIT press.

Dong, X., Zhang, C., Ge, Y., Mao, Y., Gao, Y., Lin, J., Lou, D., et al. (2023). C3: Zero-shot text-to-sql with chatgpt. *arXiv preprint arXiv:2307.07306*.

Dunn, A., Dagdelen, J., Walker, N., Lee, S., Rosen, A. S., Ceder, G., Persson, K., & Jain, A. (2022). Structured information extraction from complex scientific text with fine-tuned large language models. *arXiv preprint arXiv:2212.05238*.

Dyachenko, Y., Nenkov, N., Petrova, M., Skarga-Bandurova, I., & Soloviov, O. (2018). Approaches to cognitive architecture of autonomous intelligent agent. *Biologically Inspired Cognitive Architectures*, *26*, 130–135. https://doi.org/https://doi.org/10.1016/j.bica.2018.10.004

Finin, T., Fritzson, R., McKay, D., & McEntire, R. (1994). KQML as an agent communication language. *Proceedings of the Third International Conference on Information and Knowledge Management*, 456–463. https://doi.org/10.1145/191246.191322

Franklin, S., & Graesser, A. (1996). Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *International workshop on agent theories, architectures, and languages*, 21–35.

Frey, C. B., & Osborne, M. A. (2017). The future of employment: How susceptible are jobs to computerisation? *Technological Forecasting and Social Change*, *114*, 254–280. https://doi.org/https://doi.org/10.1016/j.techfore.2016.08.019

Gokturk, B. (2024, April). Announcing Vertex AI Agent Builder: Helping developers easily build and deploy gen AI experiences. https://cloud.google.com/blog/products/ai-machine-learning/build-generative-ai-experiences-with-vertex-ai-agent-builder

Haerem, T., Pentland, B. T., & Miller, K. D. (2015). Task Complexity: Extending a Core Concept. *Academy of Management Review*, *40*(3), 446–460. https://doi.org/10.5465/amr.2013.0350

Heider, F., & Simmel, M. (1944). An experimental study of apparent behavior. *The American journal of psychology*, *57*(2), 243–259.

Herbold, S., Hautli-Janisz, A., Heuer, U., Kikteva, Z., & Trautsch, A. (2023). A large-scale comparison of human-written versus ChatGPT-generated essays. *Scientific Reports*, *13*(1), 18617.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

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*.

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., et al. (2023). Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Huang, J., & Chang, K. C.-C. (2022). Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.*, *55*(12). https://doi.org/10.1145/3571730

Johri, P., Khatri, S. K., Al-Taani, A. T., Sabharwal, M., Suvanov, S., & Kumar, A. (2021). Natural Language Processing: History, Evolution, Application, and Future Work. In A. Abraham, O. Castillo, & D. Virmani (Eds.), *Proceedings of 3rd international conference on computing informatics and networks* (pp. 365–375). Springer Singapore.

Joseph, S. R., Hlomani, H., Letsholo, K., Kaniwa, F., & Sedimo, K. (2016). Natural language processing: A review. *International Journal of Research in Engineering and Applied Sciences*, *6*(3), 207–210.

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

Kandpal, N., Deng, H., Roberts, A., Wallace, E., & Raffel, C. (2023). Large language models struggle to learn long-tail knowledge. *International Conference on Machine Learning*, 15696–15707.

Keles, F. D., Wijewardena, P. M., & Hegde, C. (2023). On the computational complexity of self-attention. *International Conference on Algorithmic Learning Theory*, 597–619.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., & Sabharwal, A. (2022). Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

Kim, H., So, B.-H., Han, W.-S., & Lee, H. (2020). Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment*, *13*(10), 1737–1750.

Kojima, T., Gu, S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large Language Models are Zero-Shot Reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D.

Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (pp. 22199–22213, Vol. 35). Curran Associates, Inc.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-t., Fried, D., Wang, S., & Yu, T. (2023). DS-1000: A natural and reliable benchmark for data science code generation. *International Conference on Machine Learning*, 18319–18345.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., et al. (2024). Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, *36*.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., et al. (2022). Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Liddy, E. D. (2001). Natural language processing. In *Encyclopedia of library and information science* (2nd). Marcel Decker, Inc.

Liu, P., & Li, Z. (2012). Task complexity: A review and conceptualization framework. *International Journal of Industrial Ergonomics*, *42*(6), 553–568. https://doi.org/https://doi.org/10.1016/j.ergon.2012.09.001

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2021). Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *arXiv preprint arXiv:2107.13586*.

Liu, Y., Cao, J., Liu, C., Ding, K., & Jin, L. (2024). Datasets for Large Language Models: A Comprehensive Survey. *arXiv preprint arXiv:2402.18041*.

Luck, M., & d'Inverno, M. (2001). A conceptual framework for agent definition and development. *The Computer Journal*, *44*(1), 1–20.

Ludvigsen, K. G. A. (2023, March). ChatGPTs electricity consumption, pt. II. https://kaspergroesludvigsen.medium.com/chatgpts-electricity-consumption-pt-ii-225e7e43f22b

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. *Proceedings of the Python in Science Conference*, *Vol. 445*, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

Miyazaki, K., & Sato, R. (2018). Analyses of the Technological Accumulation over the 2 nd and the 3 rd AI Boom and the Issues Related to AI Adoption by Firms. *2018 Portland International Conference on Management of Engineering and Technology (PICMET)*, 1–7.

Nakabayashi, A., & Wada, H. (2016). *Changes in Data Analysis Technology in the Manufacturing Industry in the Midst of the Third Artificial Intelligence Boom* (tech. rep.). Yokogawa Technical Report English Edition.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al. (2021). Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

OpenAI. (2023a). GPT-4 Technical Report. *ArXiv*, *abs/2303.08774*. https://api.semanticscholar.org/CorpusID:257532815

OpenAI. (2023b, March). GPT-4. https://openai.com/index/gpt-4-research/

OpenAI. (2024). Code Interpreter - OpenAI API. https://platform.openai.com/docs/assistants/tools/code-interpreter

pandas development team, T. (2024, April). *Pandas-dev/pandas: Pandas* (Version v2.2.2). Zenodo. https://doi.org/10.5281/zenodo.10957263

Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., & Dean, J. (2021). Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*.

Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A. H., & Riedel, S. (2019). Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

Popescu, A.-M., Etzioni, O., & Kautz, H. (2003). Towards a theory of natural language interfaces to databases. *Proceedings of the 8th International Conference on Intelligent User Interfaces*, 149–157. https://doi.org/10.1145/604045.604070

Pourreza, M., & Rafiei, D. (2023). DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in neural information processing systems* (pp. 36339–36348, Vol. 36). Curran Associates, Inc.

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*.

Radford, A., & Narasimhan, K. (2018). Improving Language Understanding by Generative Pre-Training. *OpenAI*. https://api.semanticscholar.org/CorpusID:49313245

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. (2021). Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, *21*(1), 5485–5551.

Roberts, A., Raffel, C., & Shazeer, N. (2020, November). How Much Knowledge Can You Pack Into the Parameters of a Language Model? In B. Webber, T. Cohn, Y. He, & Y. Liu (Eds.), *Proceedings of the 2020 conference on empirical methods in natural language processing (emnlp)* (pp. 5418–5426). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.emnlp-main.437

Ruane, E., Birhane, A., & Ventresque, A. (2019). Conversational AI: Social and Ethical Considerations. *AICS*, 104–115.

Russell, S., & Norvig, P. (2021). *Artificial intelligence: A Modern Approach, Global Edition* (4th ed.).

Shanahan, M. (2024). Talking about large language models. *Communications of the ACM*, *67*(2), 68–79.

Shen, L., Shen, E., Luo, Y., Yang, X., Hu, X., Zhang, X., Tai, Z., & Wang, J. (2022). Towards natural language interfaces for data visualization: A survey. *IEEE transactions on visualization and computer graphics*, *29*(6), 3121–3144.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K. R., & Yao, S. (2023). Reflexion: language agents with verbal reinforcement learning. *Thirty-seventh Conference on Neural Information Processing Systems*.

Shoham, Y. (1993). Agent-oriented programming. *Artificial intelligence*, *60*(1), 51–92.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al. (2022). Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 27). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

Team, L. D. (2023). Lagent: InternLM a lightweight open-source framework that allows users to efficiently build large language model(llm)-based agents [(Accessed on 2024-02-16)]. https://github.com/InternLM/lagent

Topi, H., Valacich, J. S., & Hoffer, J. A. (2005). The effects of task complexity and time availability limitations on human performance in database query tasks. *International Journal of Human-Computer Studies*, *62*(3), 349–379. https://doi.org/https://doi.org/10.1016/j.ijhcs.2004.10.003

Valmeekam, K., Marquez, M., Sreedharan, S., & Kambhampati, S. (2023). On the Planning Abilities of Large Language Models - A Critical Investigation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in neural information processing systems* (pp. 75993–76005, Vol. 36). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2023/file/efb2072a358cefb75886a315a6fcf880-Paper-Conference.pdf

Van Noorden, R., Maher, B., & Nuzzo, R. (2014, October). The top 100 papers. https://www.nature.com/news/the-top-100-papers-1.16224

Varela, F. J., Rosch, E., & Thompson, E. (1991). The Embodied Mind. *The embodied mind: Cognitive science and human experience*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser,. u., & Polosukhin, I. (2017). Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 30). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2019). Superglue: A stickier benchmark for general-purpose

language understanding systems. *Advances in neural information processing systems*, *32*.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. (2023). A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Wang, Y., & Chiew, V. (2010). On the cognitive process of human problem solving. *Cognitive Systems Research*, *11*(1), 81–92. https://doi.org/https://doi.org/10.1016/j.cogsys.2008.08.003

Warren, D. H. D., & Pereira, F. C. N. (1982). An efficient easily adaptable system for interpreting natural language queries. *American journal of computational linguistics*, *8*(3-4), 110–122.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. (2022). Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter brian, b., Xia, F., Chi, E., Le, Q. V., & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (pp. 24824–24837, Vol. 35). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P.-S., Cheng, M., Glaese, M., Balle, B., Kasirzadeh, A., et al. (2021). Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359*.

Wood, R. E. (1986). Task complexity: Definition of the construct. *Organizational Behavior and Human Decision Processes*, *37*(1), 60–82. https://doi.org/https://doi.org/10.1016/0749-5978(86)90044-0

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, *10*(2), 115–152. https://doi.org/10.1017/S0269888900008122

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., & Wang, C. (2023). Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

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*.

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10. https://doi.org/10.1145/3520312.3534862

Yang, L., Zhang, S., Qin, L., Li, Y., Wang, Y., Liu, H., Wang, J., Xie, X., & Zhang, Y. (2022). Glue-x: Evaluating natural language understanding models from an out-of-distribution generalization perspective. *arXiv preprint arXiv:2211.08073*.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 32). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Zhang, X., Yu, B., Yu, H., Lv, Y., Liu, T., Huang, F., Xu, H., & Li, Y. (2023). Wider and deeper llm networks are fairer llm evaluators. *arXiv preprint arXiv:2308.01862*.

Zhang, Z., Yao, Y., Zhang, A., Tang, X., Ma, X., He, Z., Wang, Y., Gerstein, M., Wang, R., Liu, G., & Zhao, H. (2023). Igniting Language Intelligence: The Hitchhiker's Guide From Chain-of-Thought Reasoning to Language Agents.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al. (2023). A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Zhao, Z., Lee, W. S., & Hsu, D. (2024). Large language models as commonsense knowledge for large-scale task planning. *Advances in Neural Information Processing Systems*, *36*.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*.

Zhong, V., Xiong, C., & Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., et al. (2022). Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Zhu, X., Li, J., Liu, Y., Ma, C., & Wang, W. (2023). A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*.

A

Appendix

A.1 Table analysis use-cases

1. What's the function with the most tests failed for the [Software Version] release candidate?
2. What's the function with the most relative tests failed for the [New Software Version] release candidate?
3. What are the [columns] for failed tests for [Truck ID]?
4. What are the [columns] for all latest failed tests for [Truck ID]?
5. What unique trucks have had tests performed on them for the [Software Version] release candidate and what are their [column]?
6. Which function has the highest proportion of failed tests per Release Candidate?
7. What are the [columns] for [Truck ID] in the latest test suite?
8. Report the [columns] for the NOK tests for [Truck ID] in the latest test suite.
9. Report [columns] for all NOK tests on [Truck ID] in the latest test suite.
10. Report [columns] for release candidate [Software Version] and [Function Name].
11. Report [columns], count OK/NOK for all tests for RC [Software Version].
12. For each release candidate, report the function that has the highest proportion of NOK test results.
13. For each release candidate, list the test case functions that fail the most.
14. For each release candidate, list the test case functions that have the lowest proportions of not run tests.
15. What are the top three functions that have the highest proportion of failed tests per Release Candidate?

B

Appendix

B.1 Tabular data analysis knowledge in generated language

Question: What can you tell me about tabular data analysis? If I have a question that could be answered by performing analysis on a single data sheet (like an Excel), what should I do?
Answer: Tabular data analysis involves examining and processing data that is organized in a table format, typically with rows representing individual records and columns representing different variables or attributes. Here's a structured approach to performing tabular data analysis on a single data sheet, such as an Excel file:
1. Define the Question
Identify the objective: Clearly define the question or problem you aim to solve with your analysis.
Determine the scope: Understand the limits of the analysis, including the data's range and constraints.
2. Data Collection
Gather data: Obtain the data sheet from your source, ensuring it is complete and accurate.
Understand the structure: Familiarize yourself with the columns and rows, understanding what each represents.
3. Data Cleaning
Handle missing values: Decide on a strategy for dealing with missing data (e.g., imputation, removal).
Correct errors: Look for and correct any obvious errors or inconsistencies in the data.
Standardize formats: Ensure consistency in data formats, such as dates and categorical variables.
4. Exploratory Data Analysis (EDA)
Summary statistics: Calculate basic statistics like mean, median, standard deviation, and range for numerical data.
Visualizations: Use plots (e.g., histograms, bar charts, scatter plots) to visualize distributions and relationships.
Identify patterns: Look for trends, correlations, and outliers that might affect your analysis.
5. Data Transformation
Feature engineering: Create new features or modify existing ones to better capture the information in the data.
Scaling and normalization: Adjust numerical values to a common scale if necessary (e.g., using z-scores or min-max scaling).
6. Analysis and Modeling
Choose the method: Select appropriate statistical methods or machine learning models based on the question (e.g., regression, classification, clustering).
Perform analysis: Apply the chosen methods to the data, ensuring proper validation and testing.
Interpret results: Understand and explain the outcomes of your analysis, including any model performance metrics.
7. Conclusion and Reporting
Summarize findings: Provide a clear summary of your analysis and its conclusions.
Visualize results: Use charts and graphs to present your findings effectively.
Make recommendations: Based on your analysis, suggest actionable insights or next steps.
8. Documentation and Review
Document steps: Keep detailed records of your analysis process, including any code and decisions made.
Peer review: If possible, have your analysis reviewed by others to validate the approach and findings.
Tools and Techniques
Excel: For basic data manipulation and visualization.
Python: Libraries such as Pandas, NumPy, Matplotlib, and Seaborn for more advanced analysis.
R: Another powerful tool for statistical analysis and visualization.
SQL: For querying and manipulating data stored in databases.
By following these steps, you can systematically approach tabular data analysis, ensuring thorough and accurate results. If you need specific guidance on any part of this process, feel free to ask!
Question: For each tool and technique you listed, give a concise example of how to perform a filter operation and then count values in another column.
Answer: Certainly! Here are concise examples of how to perform a filter operation and then count values in another column using the listed tools and techniques:
Excel
Filter Operation:
Select the range of your data.
Go to the Data tab and click on Filter.
Use the drop-down arrows in the column headers to filter your data based on specific criteria.
Count Values in Another Column:
Use the COUNTIF function.
Example: =COUNTIF(B:B, "YourCriteria") where B:B is the column to count values based on the filter criteria applied in another column.

Python (Pandas)
Filter Operation:

```
python
import pandas as pd


# Load data into a DataFrame
df = pd.read_excel('your_data_file.xlsx')

# Filter the DataFrame
filtered_df = df[df['ColumnName'] == 'FilterCriteria']
Count Values in Another Column:

python
value_counts = filtered_df['AnotherColumn'].value_counts()
print(value_counts)

<R CODE EXCLUDED FOR BREVITY>

sql
SELECT *
FROM your_table
WHERE ColumnName = 'FilterCriteria';
Count Values in Another Column:

sql
SELECT AnotherColumn, COUNT(*)
FROM your_table
WHERE ColumnName = 'FilterCriteria'
GROUP BY AnotherColumn;
```

These examples demonstrate basic operations for filtering data and counting values in another column using each tool or technique. If you need more detailed steps or have a specific dataset in mind, feel free to ask!

Listing 2: Transcript of dialogue with ChatGPT regarding knowledge about tabular data analysis.