Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Optimizing LLM-Powered Agents for Tabular Data Analytics: Integrating LoRA for Enhanced Quality

Bc. Mikhail Poludin

Supervisor: Ing. Jan Šedivý, CSc.
Field of study: Open Informatics - Artificial Intelligence
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Poludin Mikhail**	Personal ID number:	**492571**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Optimizing LLM-Powered Agents for Tabular Data Analytics: Integrating LoRA for Enhanced Quality

Master’s thesis title in Czech:

Optimalizace LLM agentů pro analýzu tabulkových dat: Integrace LoRA pro zvýšení kvality

Guidelines:

Investigate data analytics techniques and develop a natural language interface that empowers users to query tabular data, such as spreadsheets or comma-separated value texts, using natural language questions. The primary focus is on enabling users to conduct Exploratory Data Analysis queries to extract information like averages, minimums, maximums, correlations, trends, etc., and to create simple statistical visualizations.

The study will specifically review generative Large Language Models (LLMs) also with an emphasis on their ability to generate Python code for analytical tasks. The goal is to select the most suitable LLM and fine-tune it for a specific tabular data analytics task, utilizing the LoRA (Leveraging Pre-trained Representations for Domain Adaptation) or QLoRA fine-tuning techniques to enhance accuracy. The next step is to integrate this model into the LLM-powered Agent program, with the possibility of combining multiple LLM experts.

The results will include a dataset consisting of sample analytics questions, sheet data, and respective reference outputs to facilitate model training and evaluation.

Review methodologies for assessing accuracy and propose relevant metrics to estimate the model's improvements (or the improvements of the LLM-powered Agent in general).

During the exploration, experiment with different fine-tuning setups and parameters to achieve the best balance between compute power and accuracy.

Bibliography / sources:

1. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. 2023. "React: Synergizing reasoning and acting in language models." arXiv.org. March 10, 2023. https://arxiv.org/abs/2210.03629.
2. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. 2021. "Lora: Low-rank adaptation of large language models." arXiv.org. October 16, 2021. https://arxiv.org/abs/2106.09685.
3. Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. 2023. "Qlora: Efficient finetuning of quantized llms." arXiv.org. May 23, 2023. https://arxiv.org/abs/2305.14314.
4. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., and Synnaeve, G. 2023. "Code llama: Open foundation models for code." arXiv.org. August 25, 2023. https://arxiv.org/abs/2308.12950.
5. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. 2023. "Chain-of-thought prompting elicits reasoning in large language models." arXiv.org. January 10, 2023. https://arxiv.org/abs/2201.11903.

Name and workplace of master’s thesis supervisor:

Ing. Jan Šedivý, CSc. Big Data and Cloud Computing CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **01.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Jan Šedivý, CSc.
Supervisor’s signature

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student’s signature

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor for his guidance and support throughout this project. His enthusiasm in cutting-edge technologies greatly enhanced my research and experimentation.

I am also grateful to my fellow students who have been part of my research journey. Through our meetings, where they shared their own research findings, they helped broaden my understanding of new technologies, thereby contributing to the scope of my expertise.

Most importantly, I owe a special thank you to my family. Their constant support and the opportunities they provided were not just crucial but fundamental in enabling me to begin and ultimately complete this work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

The use of generative AI tools was conducted in accordance with the official methodological guidelines of the Czech Technical University. ChatGPT was used for grammar checking and minor text reformulation. GitHub Copilot provided assistance in writing routine code and data processing scripts.

Prague, 21 May 2024

Abstract

This thesis explores the problem of analyzing tabular data using natural language, focusing on the utilization of Large Language Models (LLMs). A comprehensive literature review addresses various aspects of LLMs, including their coding capabilities, LLM Agents, and techniques for enhancing generation quality. An LLM-based Agent program was developed and is now publicly available on GitHub, also forming the basis for the experimental part of this work. Several datasets were hand-crafted and collected to facilitate the fine-tuning aimed at enhancing the performance of small, open-source models in tabular data analysis tasks. An evaluation benchmark was created, allowing for the comparison of numerous LLM Agent configurations, including those using fine-tuned LLMs and state-of-the-art (SOTA) API-based models (i.e. Claude3 and GPT models). Fine-tuning was performed on the Code Llama 7B family of models using LoRA and QLoRA techniques, which improved the performance of the Code Llama 7B Python model from 35.3% to 60.3% on the proposed evaluation benchmark. This work demonstrates that task-specific Parameter-Efficient Fine-Tuning (PEFT) on a small dataset can significantly enhance performance of LLMs. All fine-tuning experiments were tracked using MLOps tools to ensure reproducibility. Overall, this work offers a valuable comparative review of the application of LLM-based systems and associated techniques in tabular data analysis.

Keywords: LLM, Agent, NLP, Data Analysis, TableQA, LoRA, MLOps, GPT, Claude, Llama

Supervisor: Ing. Jan Šedivý, CSc.
Prague, Czech Institute of Informatics, Robotics and Cybernetics (CIIRC)

Abstrakt

Tato práce se zabývá problematikou analýzy tabulkových dat pomocí přirozeného jazyka se zaměřením na využití velkých jazykových modelů (LLM). Obsáhlý přehled literatury se zabývá různými aspekty LLM, včetně jejich programovacích schopností, LLM agentů a technik pro zvýšení kvality. Byl vyvinut program založený na LLM agentech, který je nyní veřejně dostupný na GitHub a tvoří základ experimentální části této práce. Bylo ručně vytvořeno a shromážděno několik datových sad pro doladění (fine-tuning) zaměřené na zvýšení výkonu malých open-source modelů v úloze analýzy tabulkových dat. Byl vytvořen hodnotící benchmark, který umožnil porovnání mnoha konfigurací LLM agentů, včetně těch, které využívají doladění pomocí metod LoRA a QLoRA a nejmodernější modely s přístupem přes API (Claude3 a GPT modely). Fine-tuning na modelové řadě Code Llama 7B, zejména na modelu Code Llama 7B Python, zlepšilo jeho výkon z 35,3% na 60,3% na navrženém hodnotícím benchmarku. Tato práce ukazuje, že specifické fine-tuning doladění pro danou úlohu může na malé datové sadě významně zvýšit výkon modelů. Všechny experimenty byly sledovány pomocí nástrojů MLOps, aby byla zajištěna jejich reprodukovatelnost. Celkově tato práce nabízí cenný srovnávací přehled použití systémů založených na LLM a souvisejících s nimi technik v úloze analýzy tabulkových dat.

Klíčová slova: Velké jazykové modely, Agenti, Zpracování přirozeného jazyka, Analýza tabulkových dat, TableQA, LoRA, MLOps, GPT, Claude, Llama

Překlad názvu: Optimalizace LLM agentů pro analýzu tabulkových dat: Integrace LoRA pro zvýšení kvality

Contents

1 Introduction 1
1.1 Background and Motivation 1
1.1.1 Reflections on History of Labor Automation 1
1.1.2 Evolving Landscape of Exploratory Data Analysis 2
1.2 Objectives of This Study 3
1.2.1 Current Tendencies in Natural Language EDA 3
1.2.2 Research Goals 4
1.2.3 Thesis Structure 4

2 Literature Review and Existing Solutions 7
2.1 From Statistical NLP to First Neural Models 7
2.2 Transformer Architecture 9
2.3 Large Language Models (LLMs) 11
2.3.1 Ability to generate code: benchmarks and models 12
2.3.2 Quantization 16
2.3.3 Debugging Abilities 16
2.4 Natural Language Tabular Data Analysis 18
2.4.1 Solutions without LLMs 18
2.4.2 From Prompt Design to LLM Agents 19
2.4.3 Serialization of Tabular Data 25
2.4.4 Domain specific solutions and results 26
2.5 Fine-tuning 29
2.5.1 LoRA and QLoRA 30
2.5.2 PEFT for coding LLMs 31
2.6 Gaps In Research 32
2.6.1 Contributions of This Work to Research Gaps 33

3 System Design and Implementation 35
3.1 Tech Stack 36
3.1.1 Available resources 36
3.1.2 Used Packages 36
3.1.3 MLOps and Reproducibility 39
3.2 Architecture of the Agent System 40
3.2.1 Agent module 43
3.2.2 LLMs module 43
3.2.3 Coder LLMs module 44
3.2.4 Prompts module 45
3.2.5 Code cleaning and execution 48

4 Experiments and Results 51
4.1 Training data 51
4.2 Fine-tuning 55
4.2.1 Setup 55
4.2.2 Comparative Insights and Observations 57
4.3 Evaluation Metrics 60
4.3.1 Numeric Accuracy Benchmark 60
4.3.2 LLM as evaluator 62
4.3.3 Tagging Accuracy 62
4.3.4 Visualization Execution Ratio 63
4.4 Results 63
4.5 Proposal of Visual Evaluation 70

5 Conclusion 73
5.1 Final Notes and Future Work 74

Bibliography 75

A Demonstration of basic Code Llama 7B Instruct debugging abilities 87

B Selected prompt strategy examples 89

C Reference data visualizations produced by LLM Agents 95

Figures

2.1 Evolution of main technological trends in NLP over time. 7
2.2 How Code Llama models were fine-tuned. Taken from the original release paper [50]. Infilling-capable models are marked with the ⇄ symbol. 15
2.3 Easy and Extra Hard queries from Spider [61] - a Text2SQL dataset. . 18
2.4 A general definition of an LLM-based Agent. 20
2.5 Chain-of-Thought style prompting by showing an example. *"Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"* [76]. 22
2.6 Visualization of described prompting strategies, taken from [85]. 24
2.7 Possible table serialization techniques [58]. 26
2.8 Chat2VIS approach: prompt the LLM to generate Python code, process it and execute [96]. 27
2.9 Visualization of three main classes of PEFT methods: Addition-based, Selection-based, and Reparametrization-based. Within additive methods, we distinguish two large included groups: Adapter-like methods and Soft prompts [100]. . . 29
2.10 Comparison of Selective PEFT (left), Adaptive PEFT (middle) and Reparametrization-based PEFT, namely LoRA (right) methods. Since the merged weights would be just a new "Pre-trained weights" block, LoRA is visualized during a phase before merging. 30

3.1 Directory structure of the project. 41
3.2 Flow diagram of the implemented LLM-based agent system. 42

4.1 Smoothed out evaluation losses during adapter training on 3 pandas corpora and pandas documentation corpus. 57
4.2 Comparison of CL-7B-Python for completion training cycles. Random seed stays the same, hyperparameters are different for every run. 58
4.3 Training second LoRA adapter on top of firstly pre-trained on Pandas corpus adapter. 58
4.4 Comparison of outputs for a base CL-7B-Python model (upper) and the same model fine-tuned with LoRA (lower). The completion generation begins after the triple quotation marks ("""), which mark the end of the input prompt. 59
4.5 An example how each individual test case is highlighted in the resulting HTML report. Two Test Suites are shown and both use the same comparison function to compare to floating point numbers. 62
4.6 Comparison of selected evaluation results showing the improvements of the fine-tuned model against more performant models. 68

Tables

3.1 Three non-exclusive types of table parsing available in the current implementation. Can be used separately or together in a single prompt. 47

4.1 An example of a table that is queried in one of training instances. Four out of eight columns are displayed for convenience. 54
4.2 Evaluation of agents with base LLMs. **Q** column indicates quantization bits of a loaded LLM ('-' means full precision) and **D** tells how many debugging attempts the agent had ('-' means no debug possibility). In case of Plan-and-Solve combination of GPT + CL the quantization bits are shown for the CL. Symbol *(A)* means the annotations were added to the prompts. 64
4.3 Evaluation of agents equipped with QLoRA and LoRA fine-tuned Code Llama 7B models. Coder-only configurations and the most important results were manually selected and shown here. 66
4.4 LLM as evaluator (`gpt-3.5-turbo-1106`) comparing the ground truth answers to best performing agents' answers. The score is the amount of matching answers divided by the number of questions. 68
4.5 Ratio of generated codes that were executed successfully for every visualization query from handcrafted 64 and 787 Text2Analysis sets. . . . 69
4.6 Tagging accuracy on 250 handcrafted questions. Comparing the usage of Langchain's `ChatOpenAI` Tagging with `gpt-3.5-turbo` to the usage of a small `DeBERTa-v3-base-mnli-fever-anli` classifier model. 69

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Reflections on History of Labor Automation

The history of automation and the evolution of abstractions in the work process of humanity unfolds as an amazing journey. This journey has significantly transformed the labor landscape, improving productivity and creating new realms of possibilities across industries.

The *First Industrial Revolution*, spanning in the period from 1760 to 1820, marks a pivotal point. The invention of machinery, such as the steam engine, automated manual labor, particularly in textile manufacturing, agriculture, and mining. The invention of the assembly line revolutionized manufacturing, enabling mass production of goods, notably automobiles. Those shifts not only increased production rates but also altered the overall society dynamics, leading to further urbanization and the birth of factory work.

The mid-20th century started the advent of the digital age, with the invention of the first electronic computers in the 40s, such as ENIAC [1], and the commercial release of IBM's 650 in mid 50s [1]. In a span of less than 10 years a (by that time huge) number of 2000 systems were produced. These machines were initially designed for complex computations, such as ballistic calculations and business data processing, marking the beginning of digital automation. However, programming them required intricate understanding and manipulation of hardware, limiting their accessibility and application.

The development of higher-level programming languages, starting with Fortran, and followed by COBOL and Lisp during the late 1950s, presented new significant **abstractions** over machine code and assembly language. These languages allowed people to express computations in a more natural and readable form, significantly reducing the complexity of programming. The subsequent development of structured programming languages like C and

---

[1] `https://www.cs.odu.edu/~tkennedy/cs300/development/Public/M01-HistoryOfComputers/index.html`

object-oriented languages such as Smalltalk and C++ in the 1980s further abstracted the coding process, enabling even more complex software development and wider adoption of computer technology.

The proliferation of the internet in the 1990s transformed the global economy, enabling instantaneous communication, information dissemination, and digital commerce. Programming languages such as Java, with it's slogan "write once, run anywhere"[2], and the emergence of web technologies like HTML, JavaScript, and later Python, expanded the scope of automation from local machines to global networks. The introduction of APIs (Application Programming Interfaces) further streamlined software development, allowing applications to interact and share data seamlessly, encouraging new projects and leading to the development of many more services and platforms.

Currently, it seems like the automation landscape is undergoing yet another drastic transformation with the emergence of Large Language Models (LLMs). These models, powered by research in the field of artificial intelligence and machine learning, are set to revolutionize automation by transcending traditional programing. LLMs can understand natural language, generate code, analyze data, and provide insights with minimal human input, potentially rendering traditional programming languages and manual data analysis obsolete.

1.1.2 Evolving Landscape of Exploratory Data Analysis

Exploratory Data Analysis (EDA) has been a fundamental process in statistics and data science, enabling analysts to understand the distributions, trends, and patterns within their data. The concept of EDA was popularized by John Tukey in the 1970s [2], advocating for an approach that emphasized the important role of visual methods in analyzing data, alongside traditional statistical tests. Tukey's work laid the groundwork for a more intuitive and investigative approach to data analytics, highlighting the importance of graphical representations in discovering underlying structures, patterns, and relationships.

The popularization of computers and statistical software in the late 20th century significantly expanded the capabilities of EDA. Tools like SAS (Statistical Analysis System), also in the 70s, and later R and Python programming languages, have provided powerful platforms for data manipulation. These tools allowed for more sophisticated analyses, including the ability to handle large datasets and perform complex statistical modeling. The need for more intuitive and efficient tools has so far led to the development of advanced data visualization software and automated analysis platforms, such as Tableau, which, once again, aims to democratize data analysis by making it more accessible to a higher number of people.

---

[2] `https://www.tug.ca/articles/Volume12/V12N4/V12N4_Javier_Java.html`

As we stand on the edge of a new era in data analysis, Large Language Models (LLMs) offer unprecedented opportunities to revolutionize EDA. Using their ability to understand and generate natural language, LLMs assist in automating the data exploration process, providing insights and generating code for data manipulation and visualization with minimal input. By leveraging the natural language processing capabilities of LLMs, analysts can now interact with their data in more intuitive ways, asking complex questions and receiving answers directly, without the need for extensive coding or statistical expertise. This advancement underscores the potential of LLMs in making data analysis more efficient and accessible across various domains.

1.2 Objectives of This Study

1.2.1 Current Tendencies in Natural Language EDA

The research towards automating data analytics with LLMs, with tabular data in particular, confronts several significant challenges. Predominantly, current methodologies exhibit limitations in language model's hallucination, non-determinism, autonomous reasoning and ensuring accuracy, all of which severely impact the quality of insights derived from automated data analysis. Ensuring security and safety of manipulating the data is a long way to improve as well.

Existing commercial solutions, such as well-known ChatGPT (OpenAI), Gemini (Google), etc., offer improved speed and performance on data analytics tasks, however, the underlying language models are private and huge, making the use of them contradictory. Private companies that strive for good natural language data analysis tools want to get their hands on a solution, that wouldn't allow their private data to be sent to the cloud via an API. Therefore, there is a need for accessible open-sourced models and methods, that can be instantiated on-premise. Progress in this direction would allow more independent people and companies to conduct data analysis faster, safer and cheaper.

The recent advancements in data analysis using LLM-based agents, particularly in 2022-2023, have left this area largely unexplored. The absence of comprehensive benchmarks and datasets for evaluating autonomous analysts complicates the assessment of new systems. Consequently, comparing autonomous agents with one another, especially in tasks requiring visualization creation, remains a significant challenge.

1.2.2 Research Goals

This study aims to advance the described field of data analytics through the development of a natural language interface, facilitating the direct querying of tabular data files using natural language questions. The primary objective revolves around empowering users with a simple LLM-based agent to perform data analysis efficiently, enabling them to derive insights as well as to generate simple statistical visualizations.

Natural Language Processing and LLM Optimization. A key focus is on investigating and deploying generative large language models to produce **Python code** for given analytical tasks. This includes electing appropriate LLMs for tabular data analytics tasks, and **fine-tuning** those using advanced techniques such as LoRA or QLoRA, aiming to improve accuracy and efficiency of retrieved answers. The integration of the optimized LLM into an **LLM-powered Agent** program is also ensured. Experiments with combining the strengths of different LLMs to provide a more robust solution for data analytics are conducted.

Dataset Contribution. Another output of this research will be the creation and sharing of a comprehensive **dataset**, consisting of sample analytics questions, sheet data, and corresponding reference outputs, including generated code. This dataset will serve not only as a foundation for model training and evaluation but also as a valuable contribution to the open-source community, addressing the current shortage of datasets for advanced tabular data analysis.

Methodology and Metrics for Accuracy Assessment. The research systematically reviews methodologies for measuring the accuracy of the researched and optimized models and proposes a set of metrics to evaluate the improvements brought by the assembly of the LLM-powered agent. This involves a detailed examination of the model's performance, focusing on precision and the ability to reduce mistakes in conducted data analysis, as well as the time and resource consumption.

1.2.3 Thesis Structure

This is a brief guide to what each chapter of this work contains and where the general research goals are approached:

- **Literature Review and Existing Solutions** delves into the transition from statistical NLP to the first neural models, introduces the Transformer architecture, and discusses the current state of LLMs, particularly their code generation, quantization, and debugging abilities. It further explores solutions for natural language tabular data analysis and identifies gaps in current research, positioning this work within that context.

- **System Design and Implementation** describes the technical stack used and the architecture of the Agent system developed. It covers the modular components of the system, including the agent, LLMs, and code manipulation modules, and emphasizes MLOps practices for reproducibility.
- **Experiments and Results** presents the process of the training data creation, fine-tuning experiments, and evaluation metrics used to assess the performance of the developed systems. After that, it comparatively discusses the performances of different LLM-based Agent systems that were obtained.

Chapter 2

Literature Review and Existing Solutions

2.1 From Statistical NLP to First Neural Models

This section very briefly describes the core principles and historical development of Natural Language Processing (NLP), highlighting key milestones and foundational techniques that have shaped the field. It covers the evolution from rule-based and statistical approaches to the early use of machine learning in language tasks. This background is provided for context, as the primary emphasis of this work is on contemporary high-level abstractions rather than the statistical underpinnings of early language modeling approaches.

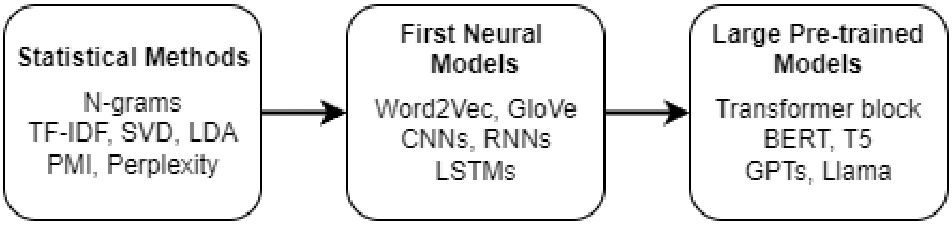


Figure 2.1: Evolution of main technological trends in NLP over time.

Probabilistic models serve as the cornerstone for understanding and predicting linguistic patterns in NLP. These models assign probabilities to sequences of words, allowing for predictions of word occurrences and facilitating tasks such as text classification and generation. To create the simplest probabilistic model one would need to know the notion of an *N-gram model*, which assigns a probability of a certain next token (could be a letter, a syllable, a word, etc.) to appear after N previously given tokens. An N-gram of size 1 is referred to as a *"uni-gram"*, size 2 is a *"bi-gram"*, and so on. The process of N-gram language model creation involves counting the occurrences of N-grams within the corpus and using these counts to estimate the likelihood of a given token following a sequence of $N-1$ tokens. For instance, the probability of the word *"Then"* following the bi-gram *"Now and"* in a tri-gram model would be calculated based on how frequently the sequence *"Now and"* is followed by *"Then"* in the training corpus, Ex. 2.1.

$$P(Then \mid Now\ and) = \frac{Count(Now\ and\ Then)}{Count(Now\ and)}. \tag{2.1}$$

One fundamental concept in evaluating the performance of these models is *perplexity*, a measure that quantifies how well a model predicts a sample. Lower perplexity indicates that the model is better at predicting the sample, making it a crucial metric for assessing model quality. It is an inverse probability of the unseen test set normalized by the number of words, Ex. 2.2.

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_N)}} \in \langle 1, \infty). \tag{2.2}$$

In the realm of text classification, Bayes' rule emerges as a pivotal foundation, enabling the categorization of text into predefined groups based on the learned probabilities. This rule applies the principles of probability to text data, offering a statistical basis for classifying text according to its content.

The evolution of NLP models introduced vector-based representations, notably through *embeddings*. Embeddings map discrete words into continuous vector spaces, enabling the computation of semantic similarity between words. Simplest techniques to generate embeddings include using term-document matrices, Term Frequency-Inverse Document Frequency (TF-IDF), and Singular Value Decomposition (SVD). While effective, these methods face challenges, such as the high computational cost of SVD for large datasets and the limitations of TF-IDF in capturing the full semantic context of words. Similarity metrics for embeddings, like cosine similarity and dot product, address these issues by quantifying the closeness between vectors, enhancing the model's ability to understand semantic relationships between words.

Despite their advancements, simple statistical NLP methods encounter limitations, such as handling the sparsity of data and capturing long-range dependencies. Researchers have developed several solutions and workarounds, including smoothing techniques for sparse data and probabilistic models for better context capture, paving the way for more sophisticated models.

The research evolution in NLP has been marked by the introduction of simple neural networks, such as word2vec and Global Vectors for Word Representation (GloVe), Recurrent Neural Networks (RNNs), and Long Short-Term Memory networks (LSTMs), addressing traditional statistical methods' challenges. Word2vec and GloVe have been instrumental in the advancements of embeddings, while Convolutional Neural Networks (CNNs), RNNs, and LSTMs have contributed to the development of language models capable of understanding sequential data, context, and temporal dependencies. These innovations tried to tackle the problems such as short context windows, the inability to capture long-range dependencies within the text, and the challenges associated with understanding the nuanced meanings of words in varying contexts. By leveraging the strengths of deep learning researchers were able to create models that not only learned word representations more effectively but also improved the accuracy of tasks like text classification, sentiment analysis, and machine translation.

2.2 Transformer Architecture

"Attention Is All You Need" [3], a research paper released in 2017, has become incredibly popular, with more than 120 thousand citations to date. This is because the new way of building neural networks introduced in this paper has led to big changes not just in NLP but also in many other areas like Computer Vision (CV) [4, 5, 6], Drug Discovery [7], Multimodal AI [8, 9], and more. During the writing process, the authors didn't expect it to have such a big impact, that's why they kept the paper consise and straightforward, focusing on the theoretical aspects and their experiments.

The smart aspect of the new network design lies in its adoption of the attention mechanism. This innovative feature enables the network to selectively concentrate on various positions of the input tokens, significantly improving its ability to grasp the overall meaning. By weighing the importance of each input token differently, the attention mechanism ensures that the model pays "attention" to the most relevant parts of the input as needed for the task at hand, whether it's understanding context, detecting relationships between words, or capturing nuances in language. This focused approach allows for a deeper and more nuanced understanding of the input data.

In a Transformer block, the attention mechanism is mixed with fully connected layers in both the encoder and the decoder parts. Sometimes, people tweak the original Transformer design for specific tasks. For example, if you're not trying to translate text but just create embeddings from it (e.g. ERNIE[10] model) or classify it (ELECTRA[11]), you might only use the encoder part (encoder-only models). On the other hand, the decoder-only models are used for plain auto regressive text generation.

Cosine positional encodings are a crucial component in the transformer architecture, enabling the model to understand the order of input tokens, which is vital for processing sequences of data like text. Unlike sequential and recurrent models, transformers treat input data as sets and, therefore, require a method to capture positional information. Cosine positional encodings introduce this information by adding vectors whose elements are computed using sine and cosine functions of different frequencies to the input embeddings. This method allows each position to have a unique encoding, yet maintains a consistent relationship between positions.

Overall, when comparing the original Transformer to previous neural network architectures for natural language modeling, quite a few benefits stand out:

- It has better focus; it is better at dealing with distant parts of the input text: Traditional methods like RNNs and LSTMs go through a sentence one word at a time, which makes it hard to link parts that are really far apart. Transformers "look" at the whole sentence at once, making this easier.

- It is faster and more efficient: Because they process all parts of the sentence at the same time, Transformers can be trained faster, especially with the help of modern GPUs.

- Simpler design: Even though they're very powerful, Transformers are actually simpler than older methods. This makes them easier to work with and integrate.

Even though the 'standard' Transformer architecture underpins every language model nowadays, it is still evolving continuously. Developers are actively working on creating innovations and refinements to boost its performance and efficiency. Here are a few examples of significant improvements, employed for instance in the original Llama models:

- Different modifications of attention mechanisms [12], such as employing a singular Key-Value (K-V) pair matrix for a group of Query (Q) matrices, primarily optimize inference. Additionally, various strategies have been developed to reduce the quadratic complexity associated with attention calculations.

- Traditional sinusoidal positional encoding has given way to Rotation-based Positional Encoding (RoPE) [13] and Positional Interpolation (PI) [14], which adjust token embeddings based on their position through a rotation mechanism. This innovation has proven effective to enable the model to handle vastly expanded context windows.

- The transition from post layer normalization to pre layer normalization has boosted convergence stability [15]. In this new approach, embeddings directly proceed through decoder blocks, with contributions from the feed-forward layers and attention mechanisms being integrated subsequently.

- The switch from the ReLU activation function to other types, like SwiGLU [16], a type of Gated Linear Unit, introduces an additional level of control over the signal flow through element-wise multiplication. This change slightly improves performance across various tasks.

- Replacing regular Layer Normalization with RMSNorm [17] simplified computations without compromising on performance, maintaining operational quality while reducing complexity.

2.3 Large Language Models (LLMs)

Following the introduction of the Transformer architecture, researchers began to explore the effects of increasing the model size. The idea that greater GPU computational capacity, when utilized on larger datasets, would enhance the performance of language models, was once again validated.

In 2018, BERT [18], encoder-only model was released in two sizes: BERT_{BASE} and BERT_{LARGE}, with the bigger one having 340 million parameters. According to the paper, BERT was pre-trained on two corpora, one of which being the English Wikipedia (2,500M words). The model noticeably outperformed previous State-Of-The-Art (SOTA) results on several NLP problems, including important benchmarks, such as GLUE [19], SQuAD [20] and SWAG [21]. The simplicity of usage and modifications of BERT paved the way for numerous research papers introducing new BERT inspired models. The model of this size was considered a Large Language Model (LLM) at that time, however, there is no specific threshold of size from which the neural network is considered large, thus making original BERT a really small language model nowadays.

The introduction of GPT-1 (Generative Pre-trained Transformer) by OpenAI in 2018 represented an advancement in the domain of text generation, demonstrating the substantial benefits of comprehensive semi-supervised pretraining followed by task-specific fine-tuning on enhancing performance across a wide array of text-based applications [22, 23]. Subsequently, OpenAI's development of GPT-2 further expanded the model itself, adding more Transformer blocks. GPT-2 model has 1.5 billion parameters, which allows for compression [24] of much more information; the training dataset for GPT-2 contained 40GB of text. It's size and size of the training corpus made the model stand on the SOTA stage of numerous benchmarks. The source code and weights are public and those are the last weights to date released by OpenAI to the public.

With the paper *"Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer"* [25] by Google, submitted in 2019, the encoder-decoder model - T5 (**T**ext-**t**o-**T**ext **T**ransfer **T**ransformer; name comes from the term *Transfer learning*) was released in several sizes. The 3 billion parameter model (3B) did achieve SOTA results on some benchmarks, but the biggest - 11B variant really showed the importance of scaling the number of parameters, beating previous best models.

By 2020, the presentation of GPT-3 with a paper *"Language Models are Few-Shot Learners"*[26] positioned it as the most advanced language model to date, distinguished by its exceptional abilities in translation, question-answering, instruction following, etc. The model possesses 175 billion parameters, making it at least ten times larger than any prior language model. In the paper, OpenAI discusses the concept of *meta-learning*, that includes the term *few-shot* *learning*, stated in the title. They research how much does it help the LLM to show it one or multiple examples of the task during the inference.

On November 30, 2022, OpenAI introduced ChatGPT, a groundbreaking application that achieved a remarkable milestone by attracting 1 million users within just five days of its release. This rapid adoption not only highlighted ChatGPT's immediate impact and the public's growing interest in advanced conversational AI but also underscored the vast potential of large language models in various applications. The app's success served as a catalyst, prompting other companies to accelerate their efforts in developing their own LLMs to compete in the rapidly growing AI market. In the wake of ChatGPT's success, OpenAI continued to innovate in the field by releasing GPT-3.5, which was trained using Reinforcement Learning from Human Feedback (RLHF) [27]. This technique involved training the model based on preferences and corrections derived from human feedback, allowing GPT-3.5 to achieve higher levels of accuracy, relevance, and safety in its outputs.

Meta's unveiling of LLaMA [28], which offers a spectrum of LLMs ranging from 7B to 65B parameters, tries to demonstrate their dedication to open science and the democratization of AI research. By providing a powerful yet more accessible tool, LLaMA caters to a wide research community. These efficient, high-performance models demand fewer computational resources, allowing for the exploration of foundational models on a broader scale. Trained on an extensive dataset of 1.4 trillion tokens across 20 major languages, LLaMA is released under a noncommercial license for research, making a significant pre-trained LLM available to the public, including academic circles.

Afterwards, numerous notable models have emerged, including GPT-4 with its iterations, LLaMA 2 models [29], and Mixtral 8x7b [30]—comprising eight 7B parameter feedforward modules, with two selected for each token based on the Mixture-Of-Experts concept. Additionally, several proprietary Claude models from Anthropic [31], Vicuna—an open-source, straightforward, and adaptable model [32], among others, have made their mark, showcasing the diverse approaches to advancing LLM technology.

2.3.1 Ability to generate code: benchmarks and models

In addition to general-purpose text-generating LLMs, this research extensively utilizes LLMs specialized in code generation. Such code-generating LLMs have gained significant attention in recent years, paralleling the interest in chat-based LLMs, due to their potential to noticeably enhance programmers' productivity and efficiency. So-called LLM-based coding assistants started to emerge. Among the most prominent of the programs that utilize LLMs to help people write code are ChatGPT, GitHub Copilot, and Amazon CodeWhisperer [33]. Programmers typically engage with these coding assistants in one of two ways: they either have a clear sequence of steps in mind and use the assistant to streamline the coding process, or they seek guidance on unfamiliar problems, leveraging the assistant as a source of suggestions for various solutions [34]. The ability to write more complex code also seems to be really useful when developing autonomous LLM agents.

While choosing the main base coding model for this project's experiments, numerous web pages and papers were looked at, so let us briefly review the best coders up to date. Models that we are going to discuss were all selected based on the following criteria, as it makes little sense to compare all LLMs, even those that could not useful for this project, as there are too many of them:

- Performance on the popular Python code writing benchmarks,
- Size of the model (smaller → better),
- Open-sourced weights,
- Ease of integration and possibility of fine-tuning.

First of all, the **benchmarks** that show how models perform on specific tasks. The original HumanEval by OpenAI [35] is a standard metric to measure LLM abilities to write code. Given a function signature with a docstring, that describes the problem, the LLM needs to generate the code below. Then the percentage of the right solutions is calculated as the final metric. Since the release, multiple improvements of HumanEval were released, for example HumanEval+ [36] is an expanded version, that contains 80 times more problems to be tested.

CoNaLa dataset [37] was created to test the ability to write small (even one-line) code snippets. One instance out of more than 2000 is for example: input is *"Sum of all values in a Python dict"* and the reference output should be *‘sum(d.values())‘*.

An older benchmark - APPS [38], consists of 10000 problem descriptions from trivial to complex with example outputs, and also the test cases that cover the generated code.

Two benchmarks from a single paper [39]: MBPP (Mostly Basic Programming Problems) and MathQA-Python serve as a solid baseline to test the Python generation. MathQA-Python challenges the LLM to extract relative data from longer texts to generate the right code.

DS-1000 [40] is a dataset collected from StackOverflow with data science related code snippets. Snippets contain code that uses typical libraries for data analysis, such as numpy, matplotlib, pandas; which strongly relates to the experimental part of this thesis. For the pandas library, it contains two types of tasks: completion and infilling. Even though the format of the data instances isn't well-suited for this project, this dataset is still used during the fine-tuning experiments.

MultiPL-E [41] is a benchmark containing translated HumanEval and MBPP to 18 different programming languages. Even when focusing on pure Python generation, it is useful to see how models perform in different languages, which can show how well the model can generalize the algorithmic processes. For example, one research also introduces a set of 852 'TransCoder' pairs, where one item is one programming language, and the second one is another language (C++, Python, Java) to test the ability to convert one language to another [42].

Another interesting benchmark worth mentioning, is SWE-bench [44], which tests the models for the ability to resolve GitHub issues, i.e. given issue text, the code segment containing a bug and the overall context of the project structure fix the given bug. The paper compares the capabilities of SOTA LLMs and talks about challenges in debugging the code autonomously. Debugging abilities of LLMs will be reviewed later on in this text, as they are a *"part of the crew, part of the ship"* in most LLM-agent systems nowadays.

There are also multiple **leader boards** available online, that compare LLMs specifically on the ability to write code. One such table is *Big Code Models Leaderboard* [1], which encompassed HumanEval and MultiPL-E tests. It allows to simply filter out models that don't fulfill your research intentions, and shows the models performances compared to each other in an interactive chart. It also shows which models are open-source, and whether a particular model has a proof of the result, or the authors just claim it in the release note. Another leader board is called EvalPlus [43] and it also references several other similar ones.

To start off with actual SOTA coding LLMs, the paper *"StarCoder: may the source be with you!"* introduces two LLMs: StarCoder (fine-tuned for Python) and StarCoderBase. Both having 15 billion parameters, they outperform previous best coding models on HumanEval and MBPP. A significant aspect of StarCoder's introduction is also its exceptional by that time ability to process over 8000 tokens of input, surpassing any other similar open LLM in terms of context length. Now, not even one year later, due to the invention of several positional encoding techniques, discussed briefly in the Transformer architecture section, this number seems really small. The model also has infilling capabilities, which means that the model can fill a missing code snippet into the prescribed place in the input prompt.

WizardCoder [45] is a series of coding LLMs that was trained using the Evol-Instruct [46] technique that was applied on the domain of programming. Evol-Instruct is a data generation method, which iteratively increases the complexity of a given problem via asking another LLM to modify the problem

---

[1] `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`

by adding constraints, complicating inputs, etc. This allows to create massive amounts of training data. Later, this technique serves as an inspiration for other similar methods (such as OSS-Instruct [47] that creates new data instances with references to open-source code snippets). Again, a smaller open-source LLM comes closer to the performance of the closed-source giants such as GPT family and Claude.

"Textbooks Are All You Need" [48] is a paper by Microsoft Research, where they introduce an LLM called phi-1, with a tiny amount of 1 billion parameters. They state that there are two ways to improve the quality of a language model: either by increasing the number of parameters, or by improving the quality of the training dataset. Inspired by TinyStories [49], they generate synthetic data, focusing on completion of Python code, achieving performance comparable with WizardCoder-16B on HumanEval and MBPP. It's also noteworthy that there is also a thing called *topic tree* creation of synthetic data, which was used to create a similar dataset named *Code Exercises* [2]. It is a sibling method to EvolInstruct, where a tree with various topics is generated, which increases the diversity and randomness of the synthetic data.

Code Llama [50] are 9 LLMs that were released by Meta AI as fine-tuned versions of Llama 2 models released earlier. 9 models come from a Cartesian product of 3 different sizes (7B, 13B, 70B) and 3 specializations, Fig. 2.2.

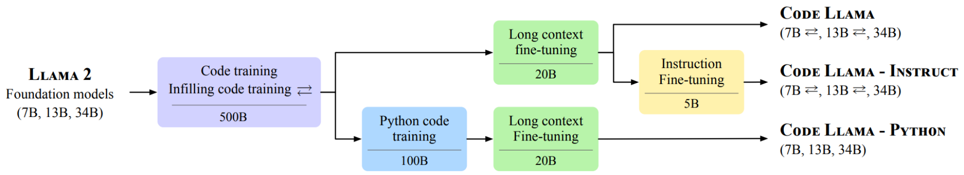


Figure 2.2: How Code Llama models were fine-tuned. Taken from the original release paper [50]. Infilling-capable models are marked with the ⇄ symbol.

This series of models offered a variety of SOTA foundational models under a permissive license, allowing both research and commercial applications. The sizes of the training corpora result in a massive knowledge within billions of parameters. 7B models can easily be run on a single GPU with VRAM of around 20GB and the 70B serves as a solid foundation for bigger projects. Here's a brief explanation of models' specializations:

- **Completion**. Similarly as any other foundational auto-regressive model, the Code Llamas can generate text completing the input text. Typical applications are completing the function body, given it's docstring description.
- **Infilling**. Causal infilling prediction allows to complete the missing code segment, that is indicated as a special token: <FILL_ME>. E.g. useful for assistants integrated to IDEs.

---

[2] `https://huggingface.co/datasets/jinaai/code_exercises`

- **Instruction** tuning allows for chat like experience with the model. While using instruction tuned Code Llama models, the input prompt should be placed in-between two tokens: [INST] and [/INST].

The simplicity of the model usage and it's further fine-tuning is also a key factor when choosing a simple, yet effective coding LLM for personal projects. It's popularity implies a lot of resources and articles with comparisons to other models and usage examples.

"Magicoder: Source Code Is All You Need" [47], which was already referenced in the WizardCoder paragraph for it's OSS-Instruct method, also introduced a novel model - MagicoderS-CL-7B, which was born by fine-tuning the Code Llama 7B Python model. Authors claim that the model outperforms the ChatGPT on HumanEval, which is amazing, keeping in mind it's size.

2.3.2 Quantization

This study will also touch on the concept of model quantization. Quantization [51] optimizes LLMs for better efficiency, especially for use on devices with limited resources. The technique converts the model's parameters from high to low precision, e.g., turning float32 with 32-bit precision into 4-bit precision, where each parameter of the model takes only 4 bits of memory. This reduction not only dramatically decreases the model's size and potentially increases its processing speed but also could lower power consumption, which is crucial for running models locally. The main quantization strategies include post-training quantization, applied after training, and quantization-aware training, integrated during the model's learning process. Although quantization can introduce errors affecting model accuracy, careful adjustment can mitigate these effects, balancing efficiency with performance.

A recent paper called *"The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits"* [52] goes to the extreme and proposes a quantized transformer in which the usual linear layer is replaced with a BitLinear layer, where weights are trained and quantized during the forward pass (the weights are divided by the mean absolute value and rounded to the nearest value from a set of $-1, 0, 1$). This model uses 20 times less energy, 3.5 times less memory at inference, and is 2.7 times faster than the fp16 model while maintaining the same quality after training on the same dataset, which envisions the potential of effective model quantization techniques in the future.

2.3.3 Debugging Abilities

The term *debugging* refers to the act of identifying and rectifying errors in specific code, often accompanied by an explanatory note. Programmers frequently seek guidance from LLMs when debugging their code, which typically includes error messages generated during compilation or execution. Given that debugging with LLMs is a common feature in numerous autonomous agent systems, including the one implemented for this project, this section provides a brief summary of various research studies on this subject.

First, a recent paper on evaluating the debugging capabilities of LLMs with a new benchmark, DebugBench (4253 instances), compares GPT models from OpenAI with Code Llama and BLOOM [53] and states that *"while closed-source models like GPT-4 exhibit inferior debugging performance compared to humans, open-source models such as Code Llama fail to attain any pass rate scores"*; reports accuracy being equal to 0 for all Code Llama family models [54]. The statement about GPT models is valid, as the GPT models show remarkable debugging capabilities. However, the claim that Code Llama models don't have internal knowledge on fixing code mistakes, as those probably weren't in the training corpora, seems biased. After examining their Appendix section with examples of used prompts and seeing an instruction-style prompt being used for the base Code Llama 34B model, which wasn't trained for instruction alignment, the bar for this research paper had fallen lower. It was decided to showcase the Code Llama 7B model's debugging abilities via a simple, non-biased demonstration, shown in Appendix A. Further evaluations with the tabular data analysis benchmark in the experiments section also show that debugging abilities are present even in 7B models and improve the performance significantly.

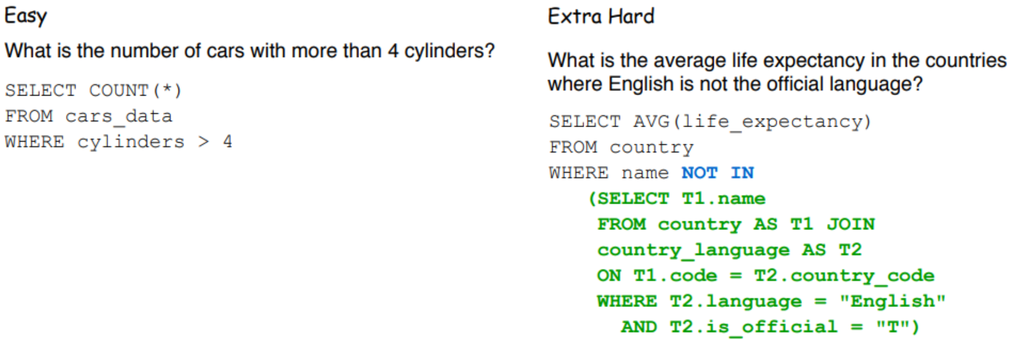
"Large Language Models Cannot Self-Correct Reasoning Yet" [55] studies the intrinsic abilities of LLMs to self-correct their initial reasoning without any other additional information. They test this on math problems as well as on common sense question answering and conclude that self-correction without outside knowledge can be beneficial, however sometimes the performance degrades after introducing self-correction.

Self-Debugging [56] is a straightforward method to correct faulty code via a cycle that repeatedly calls the LLM with additional feedback from unit tests and a few examples. This significantly helps the model, improving its scores on popular programming benchmarks. On a Text-to-SQL dataset, Spider, the unit tests are not available, but the system's capabilities still improve after the introduction of the code explanation feature.
2 *"LDB: Large Language Model Debugger via Verifying Runtime Execution Step-by-step"* [57] goes further and propose using LLMs in a IDE-style debugging, i.e. creating a branching tree of a program and executing it step by step, showing that open-source models such as StarCoder and Code Llama can improve on HumanEval even more, when compared to Self-Debugging. Both Self-Debugging and LDB fall more into the topic of autonomous LLM agents, which is discussed in the next section.

2.4 Natural Language Tabular Data Analysis

2.4.1 Solutions without LLMs

Before LLMs became popular, a challenge with performing basic operations on tables using natural language had been present for a while, here, we'll briefly focus on methods that were developed earlier. Recent work reviewing every major research in this field [58] divides table processing tasks to three groups. First, **prediction** of trends and different kinds of classifications. Some traditional approaches may include tree-based approaches, such as decision trees, simple convolutions and recurrent networks, etc. Second, there is data **generation**. This is used to augment the data or model the tabular data. And thirdly, and most importantly for this work, is **table understanding**. In other words - question answering e.g. by using filtering, doing statistical and mathematical operations, and sorting samples. This is exactly what this work focuses on, along with creating simple visualizations, which could highlight statistical findings or reveal patterns, since people often understand visual information better than text. We'll revisit these two types of tasks (statistical analysis and visualizations) several times when we talk about the experimental part of this study.



Easy

What is the number of cars with more than 4 cylinders?

```
SELECT COUNT(*)
FROM cars_data
WHERE cylinders > 4
```

Extra Hard

What is the average life expectancy in the countries where English is not the official language?

```
SELECT AVG(life_expectancy)
FROM country
WHERE name NOT IN
   (SELECT T1.name
    FROM country AS T1 JOIN
    country_language AS T2
    ON T1.code = T2.country_code
    WHERE T2.language = "English"
      AND T2.is_official = "T")
```

Figure 2.3: Easy and Extra Hard queries from Spider [61] - a Text2SQL dataset.

One major challenge in understanding tables is converting natural language requests into SQL query statements, known as Text2SQL or NL2SQL. This issue has led to various solutions, ranging from semantic parsing, which fills in missing parts of a query, to sketch-based methods using dependency graphs [59], and even employing deep neural networks with techniques like reinforcement learning. The Seq2SQL paper [60] introduced the WikiSQL dataset, a large collection of queries for over 24,000 small tables from Wikipedia, along with their corresponding SQL statements. However, most of these queries are simple, single-line SELECT statements focused mainly on selecting cells from poorly specified attribute names. This simplicity means the dataset does not pose a significant challenge to advanced LLM Agent systems or directly contribute to their enhancement without further modifications to it. Another dataset for the Text2SQL task is Spider [61]. Annotated by humans, it contains a large amount of queries, which also range from 'Easy' to 'Extra Hard' posing a challenge even for current SOTA systems, Fig. 2.3. *WikiTableQuestions* [62] is an old dataset similar to WikiSQL, which contains answers to each question in a form of a number, a string or a list of those. It was developed in 2015 to test their logical-form semantic parsing system.

As for the automatic data visualization from natural language (NL2VIS), there were also quite a few attempts based in semantic parsing and heuristic algorithms. Later, Transformer-based encoder-decoder model - ncNet [63] was implemented as a sequence-to-sequence model, that generates so-called Vega format outputs, which are essentially JSON-like objects with predefined fields specifying values to be displayed, colors, positions, etc., serving as configuration files for the resulting visualization. They also allowed passing the selected type of plot as one of the inputs to the network, which resulted in improvement in generation quality as expected. In addition, RGVisNet [64] uses a database of Vega format JSONs to automatically select similar templates for the output visualization. In other words, it outperformed the ncNet by introduction of a simple retrieval system. Some works have also addressed the evaluation of quality of generated plots. *"DeepEye: Towards Automatic Data Visualization"* [65] train several binary classifiers and ranking systems, that tell which kinds of plots would be best suited for the data.

2.4.2 From Prompt Design to LLM Agents

Since the introduction of LLMs, which in contrast to other, more traditional approaches to TableQA were finally able to tackle much more complex problems and to obey natural language instructions on a new level, the field dived into the research of LLM reasoning capabilities, or simply put the research of LLM intelligence. This resulted in several new sub-fields of Machine Learning (ML) and AI, which are a direct focus of this work. This includes **Prompt Design** problem, **Retrieval Augmented Generation (RAG)** problem, and the challenge of application construction that utilizes LLMs for human-like reasoning, i.e. **LLM-based Agents**.

As the definition of this term is vague, for this study *prompt* will be defined as a simple textual input to the LLM. It may also include special tokens - words or phrases encased in specific symbols or formatting that signal the LLM to perform certain actions or interpret the text in a particular way. These special tokens serve various purposes, such as specifying the format of the output, indicating the start and end of a segment of text for processing, or triggering a specific mode of response from the LLM (e.g. [INST] and [/INST] instruct Code Llama models to reply in instruction or chat-like manner).

Retrieval Augmented Generation (RAG) enhances language models by integrating them with external factual knowledge, thus addressing their inherent limitations in accessing specific or current information. This method increases language models' effectiveness in tasks demanding precise factual knowledge. RAG operates through a two-component system: the retriever and the generator. The retriever sources relevant facts from an external database or a knowledge base in response to an input prompt. These facts are then embedded into the input for the language model, equipping it with the necessary context to produce accurate and informed responses. *"Survey on Factuality in Large Language Models: Knowledge, Retrieval and Domain-Specificity"* [67] meticulously discusses all the details on fact-consistency and describe how and where methods of RAG can be applied. For instance, when asked to identify the best dystopian novel, a language model might choose any novel it recalls from its training data. However, using RAG, we can enhance the query by pre-loading a vector database (e.g., FAISS [68], ElasticSearch [69]) with data on the top 1000 best-sellers. This allows us to match the query with relevant entries using embedding similarity. Consequently, the model can provide a more accurate answer, such as '1984 by George Orwell,' by leveraging specific, targeted information.

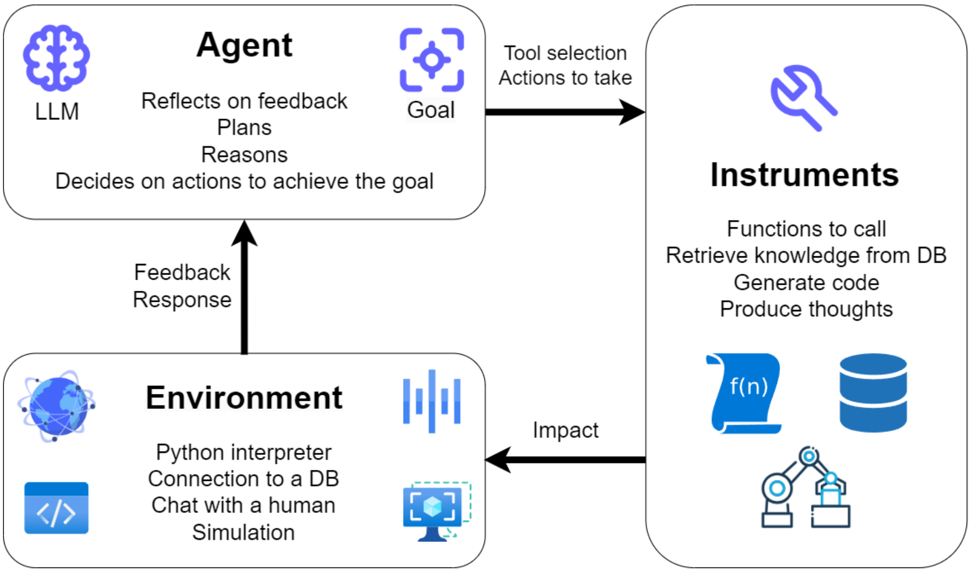


Figure 2.4: A general definition of an LLM-based Agent.

It is also crucial to define an *Agent* [70, 71]. There are many definitions of agents, but sticking to the main one, similarly to a concept of an agent in reinforcement learning, we can say that agents are systems that interact with the dynamic environment, perceive it, and act to achieve their goals or prescribed tasks, Fig. 2.4. The LLM then becomes the brain of the agent. Simply put, an LLM-based Agent is typically a "scaffolding" [3] program, that calls the language model to generate text that could in-turn give next instructions to the program. Here are some use-cases where agents operate:

- Perceiving information from the outside world, even from various sources. This information can be, for example, a response from a user, web page

---

[3] `https://metr.org/blog/2023-08-01-new-report/`

content, a database view, or the result of a program execution. Even though LLMs work with text inputs only, it is possible to modify them to accept other data types, such as images. This can be achieved with multi-modal models that encode the input image to the same vector space and then decode it into textual output [72]. In this way, pure LLM-based Agents can be transformed into embodied robots that see and act in the physical world.

- Keeping a memory. Other than the compressed training information in the model's parameters, the memory of an LLM agent can be of two types. First, there is short-term memory stored in the input prompt to the model. Given that the context windows of modern LLMs are becoming increasingly large, this can usually hold a long conversation or even several smaller books, to which an agent can refer. The second type of memory is achieved through RAG, so the agent has factual support for its answers. Another angle to look at agents is as if they were intermediate steps in the RAG pipeline. Agents are able to format the initial query, navigate the search for information in the multi-page database, and apply tools to the retrieved data. Besides information retrieval, RAG can also be utilized when an agent has a large set of available actions. It can then apply vector similarity search on the definitions and retrieve the best action to achieve its goal.

- Planning, reasoning, and reflecting back on itself through feedback. Through prompting, an agent can be instructed to plan the steps to achieve a goal, to reason, and to conclude which sequence of actions to take. Given that the text from the feedback loop usually goes to the LLM input, it can reflect on the achieved results and control its next actions based on it.

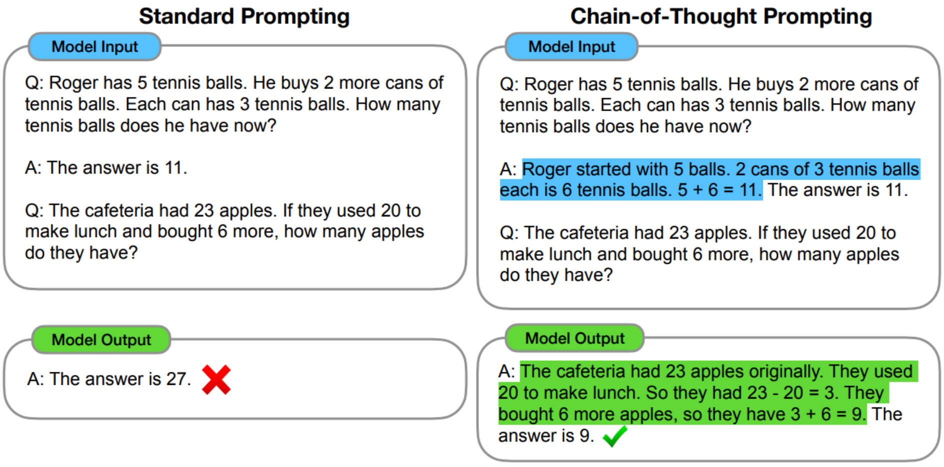
- Using instruments available in the environment. An agent typically has a set of tools that it can use. For instance, these tools can range from a Python interpreter, where the code generated by an LLM is executed, to a browser search bar, where the agent can find additional information, or even to robotics components, which are activated by simple instructions generated or selected by an LLM. The LLM can be trained to use these tools by formatting the output with special characters and specifying parameters for a selected API call [73].

- Playing a role can also be considered as a minor high-level LLM agent ability. By extensively prompting the LLM to act in a certain manner, the user could have an experience of communication with a specific profile. For example, simply telling the LLM to act as a language teacher, could redirect the feel of interaction with a user, and could even improve the response quality.

Every application utilizing LLMs faces the problem of effectively selecting prompt carcasses. Many works have already addressed this and underlined the importance of suitable prompt design. As LLMs are shown to be prompt-sensitive [74], it is usually necessary to perform prompt-tweaking tests when evaluating the final system. Such tests can show how much the system is sensitive to small and large changes in the prompt format and the overall language quality (prompt degradation), which can also examine the ability of the model to ignore harmful prompt injections.

When instructing the model to generate text in a specific format or show the model the preferred thought process, it is very useful to use so-called *In-context learning* (few-shot prompting) [75]. This means including one to several examples in the input prompt. It has been shown that LLMs improve their scores on most popular benchmarks when provided examples compared to zero-shot prompting. Studies also show that larger models exhibit more improvement when faced with a few-shot prompt compared to smaller LLMs, partially due to their limited ability to follow instructions.

Chain-of-Thought (CoT) [76, 77] is a highly discussed prompting method that was popularized due to its simplicity and effectiveness. The method instructs the model to produce its response in a style similar to a human thinking out loud. It can be achieved in multiple different ways, for example, directly telling the model to *"think step by step"*, or employing few-shot examples, Fig. 2.5. A method without changing the prompt was even proposed [78], which only alters the model's decoding process, eliminating the need for manual prompt creation.



Standard Prompting

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The answer is 27. ✗

Chain-of-Thought Prompting

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Figure 2.5: Chain-of-Thought style prompting by showing an example. *"Chain-of-Thought Prompting Elicits Reasoning in Large Language Models"* [76].

This helps the model to generate more precise answers and improves the model's accuracy on math and logical questions. As a side effect, it also makes the LLM more interpretable and comprehensible, since a human can more easily identify the reasons why the model made a certain decision. However, it was also shown that the CoT with Few-shot examples can hurt the model's performance by generating texts unsupported factual information from the context [79].

Self-Consistency [80] is a new decoding method that uses CoT prompting and samples several reasoning paths diversely instead of greedily and then marginalizes over the most consistent answers using majority voting with possible multiplication by predefined parameters. The paper shows a dramatic increase in accuracy on popular reasoning benchmarks. *Self-Discover* [81] is a more complex approach, building further on top and beating Self-Consistency on general world knowledge benchmarks.

Another method to improve the model's ability to produce the best reasoning path that converges to a correct solution is *Step-Back Prompting* [82]. It consists of two steps, which already falls into a category of simple LLM-based agents. First, the program asks the language model to generate an abstract higher-level question that relates to the original query and answer it. This helps the model to come up with additional context hint for itself, which is included to the input for the next step, actually answering the original question. The paper also discusses the results of experiments with open-source models, such as PaLM-2 and Llama-70B and presents the improvements over basic CoT and In-Context learning.

Plan-and-Solve Prompting [83] was proposed to improve the zero-shot CoT accuracy on various benchmarks. Similarly to Step-Back Prompting, this approach also consists of two trivial steps. Given a question to answer, the LLM is first prompted to generate a multistage plan and then giving the plan to the model for it to follow while answering the question. This method is also easily adapted to any reasoning tasks, not related to math or logic, such as common sense and programming. Looking ahead, it's worth noting that the method also proved to be useful in the experimental section, as we will discuss later. *Least-to-Most* prompting [84] decomposes the initial problem into smaller sub-problems, and then, sequentially solves using two different prompts, which is highly analogous to Plan-and-Solve.

Tree of Thoughts [85] and *Graph of Thoughts* [86] are methods that generalize the concept of Chain-of-Thought allowing the model to generate multiple reasoning paths, evaluate the potential of continuing that path, and select the best one at the end. Since it's usually not possible to go through all solutions, similarly to Monte Carlo Tree Search (MCTS) algorithm, these approaches constraint itself to the most perspective ones according to LLM's opinion. They dramatically beat classical CoT on very specific problems like Game of 24 or sorting an array.

Some attempts were also made to automate manual prompt creation, as it can take a lot of time for a human to properly design one. For example, *"Automatic Prompt Selection for Large Language Models"* [87], a multi-step approach, utilizes LLMs to generate prompts for each cluster of questions

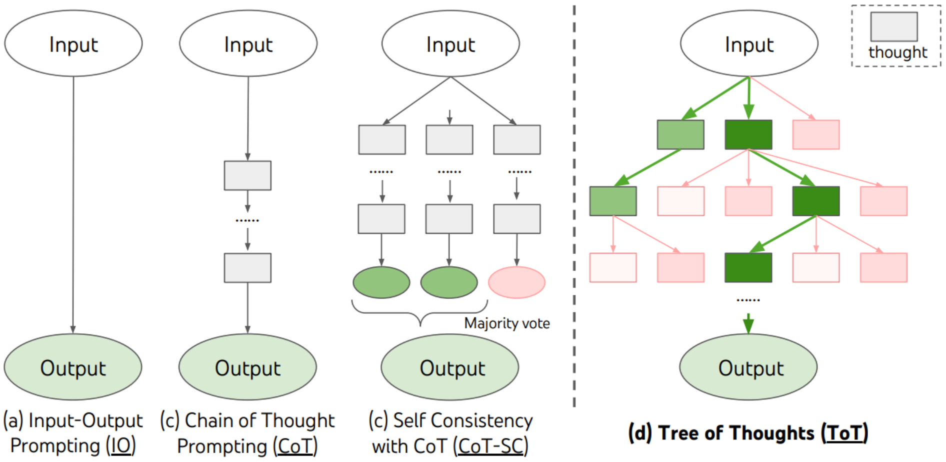

Input
Input
Input
Input
thought
......
......
......
......
Majority vote
......
Output
Output
Output
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
(c) Self Consistency with CoT (CoT-SC)
(d) Tree of Thoughts (ToT)

Figure 2.6: Visualization of described prompting strategies, taken from [85].

in the training dataset, as well as to select the best ones and evaluate them. In *"Large Language Models As Optimizers"* [88], the authors repeatedly call the LLM, each time with an updated solution and the value of an objective function for a given problem (e.g., Traveling Salesman Problem), and successfully optimize it, i.e., literally using an LLM as a very generalized numeric optimizer.

Diving deeper into the concept of an LLM Agent, *ReAct* (Reasoning + Acting) [89] is a classical, highly-cited approach that combines CoT prompting for reasoning and tool selection for acting in the given environment. It is a very general and simple framework that allows better performance on a variety of reasoning tasks (e.g., on specialized topics with fact-checking and question answering). Given the reasoning history, an agent instance enters a loop (a reasoning chain) where in each step it proposes the next action (a thought) and executes it. The main point of the method is that the agent uses the results of its actions to think about the next step. The method is currently implemented in the *Langchain* library and supports question answering with tabular data. *Reflexion* [90] is another similar approach, where instead of passing the text result of an executed action (e.g., a number), another expert LLM is utilized to "reflect" on the achieved result. This additional step helps the agent significantly and improves performance on multiple benchmarks.

The ReAct authors also propose fine-tuning the models for further improvements and discuss the possibilities of combining RL paradigms with LLM-based Agents. The case study [91] unfolds this idea and proposes a system that combines a RL agent with a language-driven one as a simple teacher and student system.

A large number of public projects without papers about LLM-Agents also exist, and one good example is *DemoGPT* [4], where a complex agent system is utilized to automate the initial creation of LLM-based applications, which could serve as a Proof-Of-Concept (POC) for the final product. It involves the utilization of multiple databases for retrieval, planning steps, code generation, code concatenation, and code testing.

MetaAgents paper [92] contains a description of tests on the ability of LLM agents to behave in a social multi-agent environment. It describes the simulations that were conducted and show promising performance along with several limitations of such systems. This shows that the memory-keeping multi-agent systems are also capable of task solving and can be utilized for specific needs. *LLM-Coordination* study [93] compares multi-agent systems to SOTA RL methods for playing 4 pure coordination games and claim that agents equipped with the GPT-4-turbo model achieve comparable results and even indicate that LLM-based agents are more stable to new unseen game partners.

There is also a significant need for a good benchmark for every researched task. *AgentBench* [94] is such a benchmark for LLMs and agents. It includes eight different tasks, such as web browsing, lateral thinking puzzles, and database operations. Therefore, this is a benchmark to consider when dealing with SQL-generating systems. However, it focuses solely on the SQL language and includes operations on multiple-table databases. The study additionally reveals that LLM systems available through commercial APIs tend to surpass their open-source counterparts in performance, pinpointing the primary challenges in developing practical LLM agents as their limited capabilities in long-term memory, decision-making, and following instructions.

2.4.3 Serialization of Tabular Data

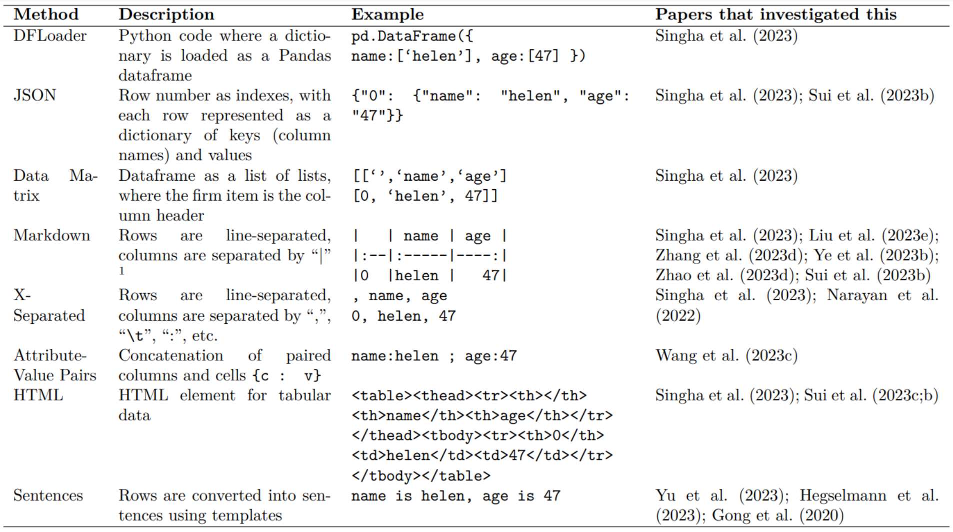
Due to the fact that the database tables can be large in size, it is needed to find a way how to serialize them into the string format for further exposure to the LLM. This section is a short condensation of mainly two big studies. One has already been mentioned, it is a recent TableQA survey paper [58] and another one is *"Table Meets LLM"* [95], which tests GPT models on understanding the serialized table that has been passed as an input to the LLM as is.

The focus of this thesis is on processing tabular data of all sizes, i.e., tables could be large in rows and columns. That's why it is important to consider an approach whose serialization would:

- Fit in the context window of most LLMs, preferably minimizing the number of input tokens used,
- Maximize the LLM's understanding and performance,
- Make the process of altering the structure and adding new information as simple as possible.

---

[4] `https://github.com/melih-unsal/DemoGPT`



Method	Description	Example	Papers that investigated this
DFLoader	Python code where a dictionary is loaded as a Pandas dataframe	`pd.DataFrame({ name:['helen'], age:[47] })`	Singha et al. (2023)
JSON	Row number as indexes, with each row represented as a dictionary of keys (column names) and values	`{"0": {"name": "helen", "age": "47"}}`	Singha et al. (2023); Sui et al. (2023b)

Data Matrix	Dataframe as a list of lists, where the firm item is the column header	`[['','name','age']`
		`[0, 'helen', 47]]`
			Singha et al. (2023)

Markdown	Rows are line-separated, columns are separated by “|”[1]	`|   | name | age |`
		`|:--|:-----|----:|`
		`|0  |helen |   47|`
			Singha et al. (2023); Liu et al. (2023e); Zhang et al. (2023d); Ye et al. (2023b); Zhao et al. (2023d); Sui et al. (2023b)

X-Separated	Rows are line-separated, columns are separated by “,”, “\t”, “:”, etc.	`, name, age`
		`0, helen, 47`
			Singha et al. (2023); Narayan et al. (2022)

Attribute-Value Pairs	Concatenation of paired columns and cells `{c : v}`	`name:helen ; age:47`	Wang et al. (2023c)

HTML	HTML element for tabular data	`<table><thead><tr><th></th>`
		`<th>name</th><th>age</th></tr>`
		`</thead><tbody><tr><th>0</th>`
		`<td>helen</td><td>47</td></tr>`
		`</tbody></table>`
			Singha et al. (2023); Sui et al. (2023c;b)

Sentences	Rows are converted into sentences using templates	`name is helen, age is 47`	Yu et al. (2023); Hegselmann et al. (2023); Gong et al. (2020)

Figure 2.7: Possible table serialization techniques [58].

The most straightforward approaches that fulfill these requirements are listed in Fig. 2.7. As the studies state, the most commonly used format is the Markdown. They indicate that JSON and DataFrame formats excel in tasks related to facts and table manipulations. On the other hand, at the cost of higher token usage, GPT models have a better grasp of HTML or XML formats for tabular QA and feature visualization. Markup languages, especially HTML, perform superiorly with GPT models in comparison to formats separated by characters like commas or tabs. This is believed to be due to the GPT models' extensive training on web data, making them more familiar with HTML and XML when it comes to processing table data.

Other approaches employ graph-based, tree-based and embedding-based methods for table serialization. While being more sophisticated, additional computation and less comprehensibility are seen as not justified for LLMs that work well with simpler methods. Training models to generate a table description is another way to tackle the problem, but it falls short because of possible hallucinations.

2.4.4 Domain specific solutions and results

All of the techniques described above can be utilized to analyze the tabular data. More existing solutions and applications of those are discussed here.

Chat2VIS [96] is a paper that was published shortly after the emergence of ChatGPT. The authors propose and test the simplest possible agent by carefully constructing prompts, generating code with an LLM, then filtering, formatting, and executing it to create a plot for a user's query on a selected table. This approach also serves as a de facto baseline for this thesis, where several more improvements are applied.

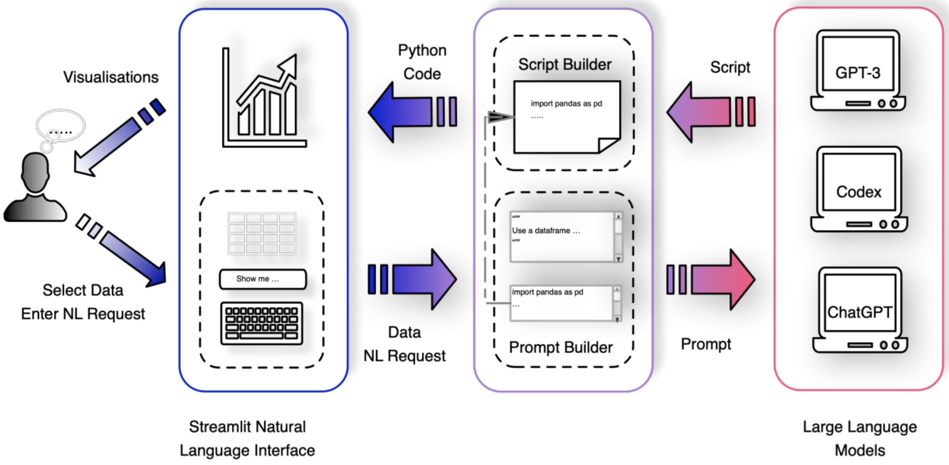


Figure 2.8: Chat2VIS approach: prompt the LLM to generate Python code, process it and execute [96].

They conduct several case studies and empirically compare results achieved with three different models (GPT-3, Codex, ChatGPT), also testing the ability to handle misspelled or ambiguous queries. However, the study lacks any numerical evaluation, which is one of the goals of this work. *"Visualization in the Era of Artificial Intelligence"* [97] expands on the same thoughts while also assessing the LLM capabilities to generate code for 2D and 3D scenes in different programming languages.

Text2Analysis [98] is an important paper for this work because it significantly contributed to the ideas of evaluating the agents for table analysis and also provided a large dataset of various types of questions for both statistical and visualization questions. To the date of writing this thesis, the dataset wasn't available on the GitHub page stated in the paper; however, the authors kindly gave me access to the pre-release version of the dataset. The dataset contains 2249 question-answer instances with the correct code. Questions are on 347 different tables. The questions ultimately fall into one of the four categories: Rudimentary Operations, Basic Insights, Forecasting, and Chart Generation. The *Rudimentary Operations* category contains queries on selecting, filtering, and performing simple aggregation operations on the tabular data. Each Rudimentary Operation query instance also has an accompanying list of operation names that are supposed to be performed, e.g., *['Pivot/groupby', 'Aggregation']*. *Basic Insights* are more difficult tasks, where the agents should know how to see trends in the data, how to detect outliers, etc. The authors state implementing 7 custom functions to get the result for each query. The *Forecasting* category is aimed at testing the ability to predict the next samples from the available data. This, however, implies generating longer code that uses other Python modules, e.g., Greykite or Prophet. The *Chart Generation* question set encompasses queries on visualizing the tabular data. The authors also state ambiguities for every task, if those are present, e.g., "Unspecified tasks" ambiguity for the query "Analyze the data." They put additional effort into making queries more difficult and concentrate on more unclear queries, where the column names are not specified directly or the task could easily be interpreted differently.

Regarding the evaluation, the authors introduce 3 metrics: Executable Code Ratio (ECR), pass@1 accuracy, and regression scores. For the pass@1 accuracy, the output of the LLM-agent must be one of the following data types: pd.DataFrame, List[int], List[float], List[str], int, float, str, dict. The comparison is conducted via a simple equality between the model's output and the reference object. Every value is transformed into a string for the purpose of comparison, encompassing all values in lists, dictionaries, and DataFrames. If the value is a number, it is rounded to two decimal places before being converted into a string. If the value is a timestamp, it is converted into a string using a single format. If the DataFrame consists of a single value, meaning it has one row and one column, this value is extracted. If there is only one column, its values are extracted as a list for additional comparisons. All these comprehensive details are crucial, as coming up with a dependable evaluation strategy was a requirement for the implementation part of this project.

The results of the evaluation of different LLMs on all 4 types of tasks are presented in detail. They show that rudimentary operations and chart generation are 2 tasks that LLMs can tackle tolerably well. But the models still struggle when presented with an unclear query where a good system would first reason on. Forecasting is the task where LLMs struggle the most as shown by all 3 metrics and especially the regression score metrics.

In November 2023, OpenAI announced and released the ability of ChatGPT to read and process files of different formats, including tabular formats: xlsx, csv. The proceses, algorithms, and prompts behind the tabular analysis are not publicly available, but as of the time of writing this text, the GPT models employed a Python interpreter to execute generated code and debug it if the result of execution wasn't as expected or produced an exception. This implies the usage of some sort of an Agent loop similar to the one implemented in this study.

There are also several public repositories available on GitHub that focus on tabular data analysis and visualization. For example: PlotAI [5] and PandasAI [6]. Both place simple LLM agent systems in a small Python package with a simple interface. This kind of project design inspired the implementation part of this work.

---

[5] `https://github.com/mljar/plotai/tree/main`

[6] `https://github.com/sinaptik-ai/pandas-ai`

2.5 Fine-tuning

Pre-trained LLMs can be further enhanced through a process known as fine-tuning. This involves training an existing base model on a specific, smaller dataset tailored to a particular task or domain of interest. By doing so, the model adapts its knowledge and capabilities to better suit the requirements of the task at hand. Fine-tuning allows for the customization of a general-purpose LLM to perform specialized functions, improving its performance on niche tasks without the need for expensive and time-consuming training a model from scratch. This is particularly beneficial in situations where data is scarce or highly specialized. The fine-tuning process essentially leverages the broad knowledge already encapsulated within the LLM, directing it towards the nuances of the specific task, leading to more accurate and contextually relevant responses.

Nevertheless, fine-tuning of a full model (i.e. modifying every learnable parameter) is still costly and requires the memory and computational power setup being the same as training the base model. That's why, numerous Parameter-Efficient Fine-Tuning (PEFT) [99] were developed and continue to be developed rapidly.

"Mini-Giants: The Triumph of 'Small' Language Models and Open Source" study [100] provides an excellent synopsis of a multitude of PEFT strategies developed up until the summer of 2023. Strategies are categorized into 3 classes: Addition-based, Selection-based, and Reparametrization-based, see Fig. 2.9.

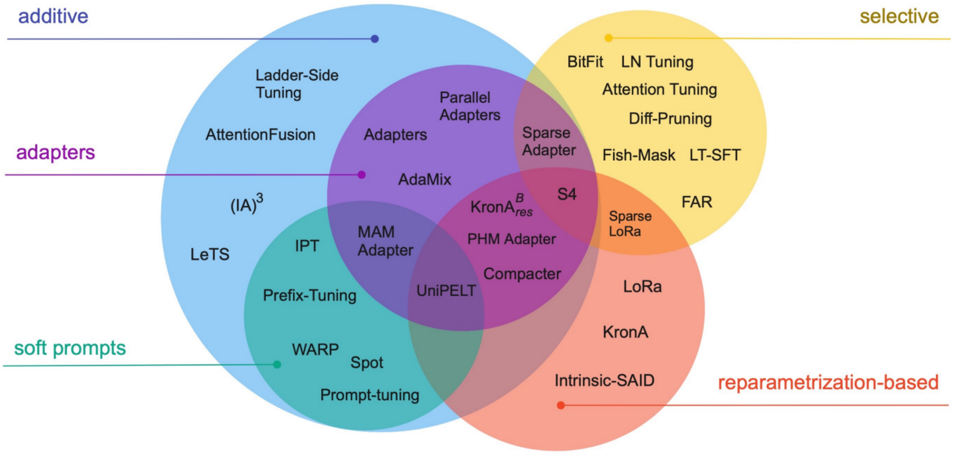


Figure 2.9: Visualization of three main classes of PEFT methods: Addition-based, Selection-based, and Reparametrization-based. Within additive methods, we distinguish two large included groups: Adapter-like methods and Soft prompts [100].

Addition-based methods introduce new parameters in addition to the initial architecture of the model and train only those. Many works focused on variations of sizes, placements and types of the introduced layers exist; where the most simple approaches insert simple linear feed-forward layers. Selective methods select a subset of parameters of the base model for the training. This subset could e.g. consist of full layers, of biases only, or even from randomly picked single parameters. Fine-tuning methods that are based on reparametrization utilize low-rank representations to reduce the count of parameters that will be trained. Likely the most recognized such method is Low-Rank Adaptation of Large Language Models (LoRA) [101].

2.5.1 LoRA and QLoRA

To improve the efficiency of fine-tuning, the LoRA technique utilizes low-rank decomposition to express weight updates with two smaller matrices, known as update matrices, Fig. 2.10. These matrices are trained to adjust to new data while maintaining a minimal number of overall changes. For the time of training the original weight matrix is preserved without additional adjustments. The final output is derived by merging both the original W and trained weights: A and B. While LoRA can be implemented on any set of weight matrices within a neural network to cut down on trainable parameters, it is often used specifically in the attention blocks of Transformer models for simplicity and increased parameter efficiency. The total number of trainable parameters in a LoRA module is influenced by the size of the low-rank update matrices, which depends on the rank r and the dimensions of the original weight matrix.

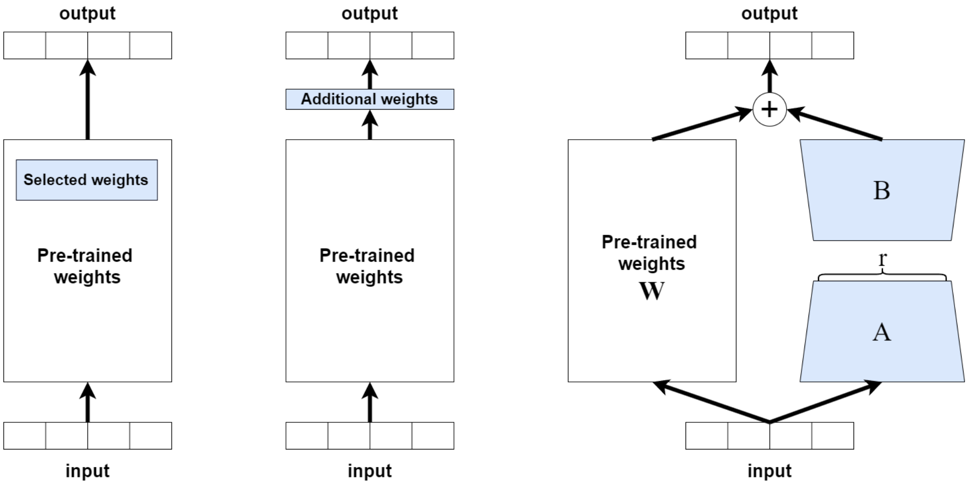


Figure 2.10: Comparison of Selective PEFT (left), Adaptive PEFT (middle) and Reparametrization-based PEFT, namely LoRA (right) methods. Since the merged weights would be just a new "Pre-trained weights" block, LoRA is visualized during a phase before merging.

LoRA is compatible with numerous other methods aimed at efficient parameter usage and can be integrated with them. There is also no increase in inference latency with LoRA, as the adapter weights can be seamlessly integrated into the base model.

QLoRA [102] method significantly advances the fine-tuning of LLMs by improving the efficiency of memory usage, making it feasible to fine-tune models up to 65 billion parameters on a single 48GB GPU while maintaining the performance standards of full 16-bit fine-tuning. QLoRA achieves this through an innovative combination of gradient backpropagation through a frozen, 4-bit quantized pre-trained weights into Low Rank Adapters. This method integrates novel techniques such as 4-bit NormalFloat (NF4), which optimizes data representation for normally distributed weights, and Double Quantization, which further reduces memory demands by quantizing the quantization constants themselves. Additionally, the method employs Paged Optimizers to efficiently handle memory spikes. QLoRA's popularity in academia and general public circles has surged due to its low memory usage and training outcomes that are on par with LoRA and full fine-tuning.

2.5.2 PEFT for coding LLMs

This small section is fully denoted to a research paper named *"Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models"* [103], which scrupulously evaluates and compares several fine-tuning techniques applied on several small ($< 1B$) and large language models in order to improve the coding abilities. They also compare fine-tuning techniques, i.e. prompt tuning, prefix tuning, IA3, full fine-tuning and LoRA with QLoRA, with In-Context learning. Tested models are CodeGen, CodeT5 and Code Llama models. The coding datasets are curated versions of CoNaLa [37] and CodeAlpaca [7]. Let us see the relevant findings of the conducted research step by step.

First, *"LLMs with PEFT consistently and significantly outperform small language models under the same GPU limit. Specifically, the best-performing LLM with PEFT surpasses the best small model by* $39.8 - 72.3\%$ *in terms of EM@k. Among different PEFT techniques, LoRA is the most effective one"*. Which means that when adapting an LLM to a domain-specific task it is usually better to invest into the model size rather than to try to conduct fine-tuning on smaller models. And then to fine-tune the selected model, LoRA is the most stable and still performs better than IA3, which was initially developed as a LoRA improvement.

In-Context Learning was also systematically outperformed by PEFT methods. However, ICL and PEFT are not exclusive and that means that can be both used to improve the performance, but the authors decided not to include this sort of experiments.

---

[7] `https://github.com/sahil280114/codealpaca`

In conclusion, the authors emphasize that QLoRA significantly cuts down on memory consumption, attaining a reduction of up to twice as much as LoRA, without compromising and even enhancing the model's performance on both CoNaLa and CodeAlpacaPy datasets. Moreover, QLoRA is able fine-tune LLMs having as many as 34B parameters, using just 24GB of GPU memory.

2.6 Gaps In Research

Significant advancements have been made in the realms of training and evaluating LLMs and general LLM-based agent systems. Yet, there remains a distinct gap in domain-specific adaptations, especially in areas like TableQA and data visualization. Methodological diversity and comprehensive comparisons of LLM capabilities in these specialized tasks are largely underexplored. The Exploratory Data Analysis (EDA) sector, in particular, presents unique challenges and intricate relationships that generic models often struggle to address effectively without tailored adjustments.

On another side, despite notable progress in proprietary commercial LLMs for code generation and tabular data analysis, there is an increasing demand for more compact, on-premise systems. Organizations are driven by the need to reduce deployment costs and the time required to integrate these systems, and to adhere to privacy regulations that restrict sharing sensitive data externally. The development of robust, standalone agent systems that do not rely on external APIs would be a significant gift for companies looking to keep their data analysis internal.

Recent research indicates that smaller language models, usually those with less than 34 billion parameters, often fail to follow instructions and generate domain-specific code as effectively as their larger or close-sourced counterparts. These smaller models frequently struggle with generalizing from prompts and show high sensitivity to the specific phrasing of prompts. Addressing these issues could involve the creation of new, specialized models or the fine-tuning of existing general models to improve their performance in specific tasks.

Moreover, there is an acute need for reliable evaluation methods for TableQA tasks. Traditional benchmarks for agent systems and methodologies for comparing agent responses are in need of refinement. Current methods, such as direct string matching and using LLMs as evaluators, are fraught with notable limitations. Developing a hybrid evaluation framework that incorporates multiple techniques could provide a more objective means of assessing the textual responses from LLM-based systems. The evaluation of generated visualizations also demands attention. The prevalent approaches, which often rely on comparing text-based plot specifications, execution ratios, and subjective human assessments, fail to adequately capture the effectiveness of data visualization tools produced by LLMs. Establishing more sophisticated evaluation metrics that can both quantitatively and qualitatively assess the quality of visual outputs is crucial for advancing the field.

2.6.1 Contributions of This Work to Research Gaps

This work directly addresses several underexplored areas identified in the research gaps, thereby making substantial contributions to the field of TableQA and data visualization with LLMs:

- **Comprehensive Comparison of Agent Systems:** This research conducts a thorough comparison of data analysis LLM-based agent systems, employing diverse metrics such as execution ratios, object comparison post-execution, and employing LLMs as evaluators. This comparison is enriched by examining the effects of multi-stage query processing, debugging, and varied prompting techniques, thus providing insights into practical implementation strategies.

- **Development of Specialized Datasets:** Recognizing the scarcity of domain-specific training and evaluation data, this work develops new handcrafted datasets tailored for both training and testing LLM agents. These datasets encompass a variety of general and visualization queries, along with corresponding answers and Python code for actual analysis. Additionally, datasets incorporating artificially generated samples from sources like Pandas documentation and filtered datasets such as OSS-Instruct and DS-1000 have been created, enhancing the diversity and representativeness of training and testing materials.

- **Exploration of Fine-Tuning Techniques:** Addressing the noted sensitivity of smaller models to prompt phrasing and their generalization challenges, this research explores and verifies hypotheses regarding Parametric Efficient Fine-Tuning techniques such as LoRA and QLoRA specifically for table analysis tasks.

Chapter 3

System Design and Implementation

The primary objective of this project was to synergize various Machine Learning (ML) tools to develop an application capable of interpreting natural language queries about selected tabular data files. The application is designed to deliver responses in various formats, including text, images, or a combination of both. To facilitate this, I developed a versatile open-source Python library. This library encompasses not just the application but also includes all auxiliary datasets and evaluation scripts employed in the experimental phase of this thesis. This library serves as a foundational framework that can be customized to meet individual requirements.

Additionally, particular attention was given to integrating the application with web platforms, where text and image responses are dispatched from endpoints. This integration had to be combined with straightforward Python usage, such as in Google Colab, where it's unnecessary to store images—they can simply be displayed using matplotlib's `show` method. Accordingly, the LLM is tasked with generating code that directly shows the visualizations, ensuring seamless operation across different environments. Moreover, more agent flow and prompt strategies can be easily added to the source code, as it is highly configurable.

It is my aspiration that both the library and this thesis will advance the field of tabular data analysis by offering an easy-to-use interface coupled with a comprehensive description and analysis documented in this text.

This chapter begins by detailing the selection of tools and frameworks that were instrumental not only in crafting the library but also in exploring various configurations of the agent and in testing and fine-tuning the internal LLMs. It then progresses to a discussion on the architecture of the package, illustrating usage examples and outlining the software design. The subsequent chapter will delve into the experimental aspects of the project, focusing on training, testing, and evaluating the agents.

3.1 Tech Stack

This section describes the tools and frameworks that are used either in the source code or were used in various training and testing pipelines. The choice of technologies was guided by their proven efficacy in machine learning tasks, compatibility with LLM architectures, and ease of integration into the Python ecosystem. In addition, the selected stack ensures that the application remains user-friendly, scalable and maintainable, making it ideal for both academic research and practical deployments. These instruments make the tech stack not only foundational but also adaptable to future changes and improvements to the code base.

3.1.1 Available resources

This project, focusing on open-source LLMs, necessitates careful management of both storage and GPU memory due to the substantial sizes of these models. For example, the smallest Llama 2 model contains a little under 7 billion parameters, which occupy approximately 14 GB. To facilitate fast inference, this model must be loaded onto a GPU. While quantizing the weights can reduce the necessary video memory, it may also lower the model's performance and potentially increase computational time.

This woek was conducted as part of the Czech Institute of Informatics, Robotics, and Cybernetics (CIIRC) research. CIIRC provided access to a computational cluster managed by the Slurm workload manager, which allowed for efficient allocation of GPU resources through simple command-line statements. For the experiments, the NVIDIA A40 graphics card with 45GB VRAM was primarily utilized for larger model inference and smaller model fine-tuning. Additionally, the NVIDIA GeForce RTX 3060, with 12GB of dedicated GPU memory, was employed for running experiments on smaller, quantized models, including the 4-bit QLoRA fine-tuning of 7B models.

3.1.2 Used Packages

Here, the libraries and dependencies that were used in the development process are briefly described.

Langchain[1] is a versatile library designed to facilitate the integration of language models with external knowledge bases and actionable systems. It primarily supports the creation and management of LLM-based agents capable of performing complex reasoning and interactive tasks. This package provides a robust framework for chaining together natural language understanding, decision-making processes, and action execution within diverse

---

[1] `https://python.langchain.com/docs/get_started/introduction`

applications. Langchain's modular architecture allows developers to easily incorporate advanced features such as reasoning chains, data retrieval, and dynamic response generation into their projects. Utilizing this library, developers can easily create LLM-based agents or use pre-existing ones, e.g. the general ReAct agent. The library undergoes rapid refactoring and development and the version compatibilities have to be controlled with additional care.

Transformers.[2] Developed by Hugging Face, the Transformers library is a comprehensive suite for NLP that provides state-of-the-art pre-trained models designed to perform a wide range of tasks. This library is instrumental in enabling quick and efficient utilization of LLMs. It simplifies the process of model loading, training, and inference, making it accessible even to those new to the ML field. It has become a standard place to store newly developed models not only for NLP, but also for other ML fields like CV. Its consistent API and easy integration with other Python libraries allow for seamless development workflows and experimentation with different models.

Datasets[3] is tightly integrated with the Transformers module, leverages concepts similar to Git and Data Version Control (DVC) for efficient dataset management. It allows users to commit and store datasets directly on the Hugging Face platform using concise Python syntax. With just a single line of code, users can upload to or load datasets from Hugging Face, streamlining the data handling process significantly. The Datasets library also features a user-friendly dataset viewer, along with straightforward mechanisms for converting dataset objects into PyTorch Dataloaders or Pandas DataFrames for easy integration into machine learning workflows. Additionally, the library supports caching mechanisms to optimize loading times and reduce redundant data processing, making it ideal for iterative machine learning tasks.

BitsAndBytes[4] is a specialized library designed to facilitate the quantization of machine learning models, enabling more efficient use of storage and computational resources.

PEFT.[5] This package, integrated with the Transformers library, simplifies the training of models and facilitates various operations on model adapters, including merging and switching. It provides an efficient approach to applying Parameter-Efficient Fine-Tuning techniques, enhancing the flexibility and scalability of model customization. Fine-tuning a model from Hugging Face platform is as straightforward as defining a Trainer object, specifying hyperparameters along with a dataset, and starting a training loop.

PyTorch.[6]In this project, PyTorch was employed in conjunction with

---

[2] `https://huggingface.co/docs/transformers/index`

[3] `https://huggingface.co/docs/datasets/index`

[4] `https://huggingface.co/docs/bitsandbytes/index`

[5] `https://huggingface.co/docs/peft/index`

[6] `https://pytorch.org/`

the Transformers library to handle various aspects of model management, including parameter loading and tokenization. This integration facilitates seamless interactions between the deep learning model architecture and the preprocessing or fine-tuning stages of development. PyTorch's flexible and dynamic nature allows for straightforward customization of neural network layers and efficient data manipulation, which is critical when working with the complex models typical in natural language processing tasks. The synergy between PyTorch and Transformers is crucial for optimizing the performance and scalability of LLMs used locally in the application.

The **OpenAI API**[7] was used to perform inference with GPT family models and to explore the capabilities of the AssistantsAPI, which supports Python REPL and internal function calling. The API provides a seamless integration, allowing for fast access to GPT models and enabling simple analysis tasks directly within the application. The API was also used to automate the datasets creation, formatting and filtering by repeatedly calling GPT models similarly to utilizing ChatGPT.

Pandas[8] and **Matplotlib**[9] are foundational libraries for this project. Pandas, with its robust and intuitive interface, serves as the backbone for handling and analyzing tabular data within the application. It allows for a variety of data manipulations that mirror SQL operations such as sorting, grouping, and filtering, in addition to offering comprehensive support for merging, joining, and time-series functionalities. Integration with libraries like NumPy and support for data serialization formats such as Pickle enhance its versatility. Matplotlib complements Pandas by providing the capability to visualize data. It enables the generation of a wide range of static, animated, and interactive visualizations, which are essential for interpreting the results of data analysis conducted by the LLMs. The integration of Pandas and Matplotlib ensures that the LLM-generated Python code can effectively process data and present insights through detailed graphical representations, solving the tasks that the user has provided.

QLoRA[10] public GitHub repository was utilized for fine-tuning LLMs. The repository was forked and modified for the needs of this project by adding additional parameters and incorporating support for the collected datasets.

llama.cpp[11][12] and **vLLM**[13] are packages designed for various operations on LLMs, including inference, quantization, and conversion of weights file types. These tools provide developers with the flexibility to optimize model

---

[7] `https://platform.openai.com/docs/api-reference`

[8] `https://pandas.pydata.org/docs/index.html`

[9] `https://matplotlib.org/stable/api/index`

[10] `https://github.com/artidoro/qlora`

[11] `https://github.com/abetlen/llama-cpp-python`

[12] `https://github.com/ggerganov/llama.cpp`

[13] `https://github.com/vllm-project/vllm`

performance across different computing environments, streamlining the deployment and scaling of LLM applications. Moreover, they offer APIs that facilitate seamless integration with existing machine learning workflows, enhancing efficiency in model management and experimentation.

Streamlit[14] is an open-source Python framework specifically designed for data scientists and AI/ML engineers. It allows users to create and deploy dynamic data-driven applications with minimal coding effort—often just a few lines. This framework streamlines the process of building interactive and visually appealing applications that can showcase data analyses, machine learning models, and more. It can quickly transform data scripts into shareable web apps, significantly reducing development time. It was used to create a minimalistic UI for the implemented agent.

The **Flask**[15] framework is a lightweight and flexible micro web framework for Python, suited for small web applications. It provides simple tools necessary to build a web service with minimal setup. During this work it was used to write several endpoints to integrate with the streamlit application.

3.1.3 MLOps and Reproducibility

In response to the growing concern over unverified research outputs, it is essential for every machine learning project to implement an MLOps pipeline. MLOps includes practices and tools that automate and streamline the lifecycle of machine learning systems from development to production and maintenance. It integrates best practices from software development and IT operations to ensure continuous integration (CI), continuous delivery (CD), and continuous monitoring of ML systems. This approach not only improves the reproducibility and reliability of models but also ensures that they remain robust and performant over time in real-world applications. By standardizing and automating model training, testing, deployment, and evaluation, MLOps facilitates faster experimentation and more consistent delivery of ML-driven solutions. It also includes version control, data tracking, and experiment logging, which are essential for tracing the evolution of models and their performance over time, ensuring that all results can be replicated and validated independently.

1. **Version control.** To track the code development process, public GitHub and private GitLab repositories[16] were created and used.
2. **PyPI** was used to share the created Python package so that it could easily be installed via a *pip install* command. **Poetry**[17] package was used as a dependency manager because it supports the uploading of the project to the PyPI.

---

[14] `https://docs.streamlit.io/`

[15] `https://flask.palletsprojects.com/en/3.0.x/`

[16] `https://github.com/poludmik/TableQA-LLMAgent/tree/master`

[17] `https://github.com/python-poetry/poetry`

3. Basic **CI/CD** process has been set up in the GitHub repository using GitHub Actions. At present, these actions are configured to test basic prompt creation due to the extensive variety of prompting strategies that require consistent verification. The setup is designed to be easily scalable to include additional tests as needed.

4. **Weights & Biases (W&B)** is both a web interface and a Python library that serves as a powerful tool for tracking experiments, visualizing data, and managing machine learning projects. It provides a centralized platform to log and compare experiments, enabling researchers to track the progress of models, understand parameter impacts, and optimize ML workflows. W&B integrates seamlessly with most machine learning frameworks (e.g., Transformers), simplifying the process of capturing hyperparameters, outputs, system metrics, and model predictions to ensure comprehensive documentation of each experiment. This integration is crucial for MLOps setups as it supports reproducibility and collaborative analysis, allowing teams to efficiently scale and iterate on machine learning projects. This work primarily used W&B to track the fine-tuning of LLMs, i.e., visualize each training run (losses, speed, learning rate curves) and store hyperparameters. **Configuration files** (.yaml) were used to simplify running experiments with a single script and to connect corresponding hyperparameters with a W&B run. This way, anyone can access detailed information about each conducted fine-tuning cycle on the page of the W&B project[18].

5. Data version control was done through both the GitHub repository for smaller files and through **Hugging Face datasets** interface. Created datasets are publicly available on my HF profile[19].

6. **Hydra**[20] is an open-source Python framework that simplifies the development of research applications. In the 'LLM as evaluator' experiments in this work, Hydra was used for managing configuration files and logging, ensuring streamlined and organized experiment setups.

3.2 Architecture of the Agent System

This section outlines the implementation details of the application designed to respond to user queries based on selected tabular data. The goal is to ensure the reproducibility of this work; therefore, the project structure and related details are thoroughly described. The system is organized as a Python package, featuring a structure typical of machine learning projects, as shown in Fig. 3.1. All development procedures are intended to be conducted from the root directory. The *main.py* script demonstrates the basic functionalities

---

[18] `https://wandb.ai/poludmik/codellama_LoRA?nw=nwuserpoludmik`
[19] `https://huggingface.co/poludmik`
[20] `https://hydra.cc/docs/intro/`

of the package.

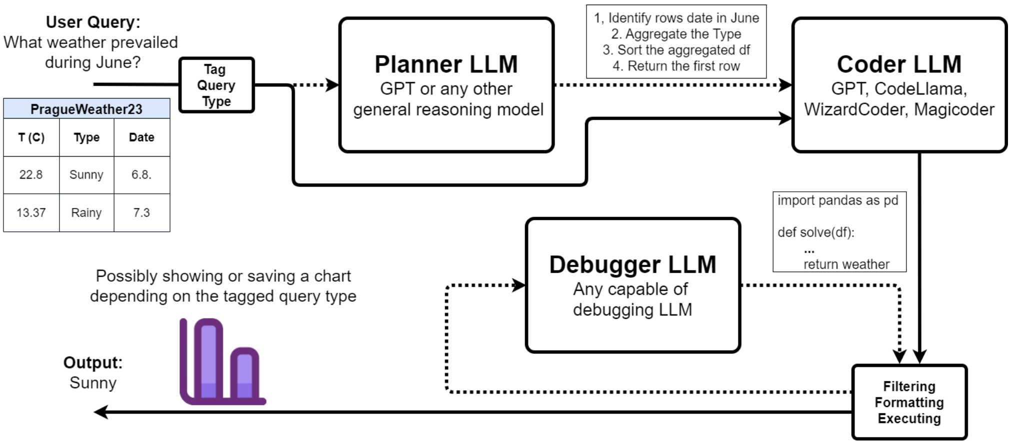
```
TableQA-LLMAgent/                                       Root directory
├── .github/
│   └── workflows/                                     CI/CD workflows
├── README/
├── tableqallmagent/                          Source code of the package
│   ├── __init__.py
│   ├── agent.py                     Constructor and the main interface
│   ├── code_manipulation.py                  Processing generated code
│   ├── coder_llms.py                    Forward passes for coding LLMs
│   ├── llms.py                            Higher level methods for LLMs
│   ├── logger.py                       Color constants for readability
│   └── prompts.py                    Prompting strategies and formatting
├── dataset/                        Multiple datasets and preprocessing
├── dist/                                                PyPI versions
├── evaluation/                                        LLM-as-evaluator
├── finetuning/
│   └── lora/                          LoRA training scripts and configs
├── plots/                          Directory to store generated images
├── tests/                                       Tests for CI/CD or pytest
├── .gitignore
├── README.md
├── main.py                                        Agent usage example
├── poetry.lock                             Poetry dependency management
└── pyproject.toml                                  Readable dependencies
```

Figure 3.1: Directory structure of the project.

At a high abstraction level, to answer a user query in natural language, an LLM must generate Python code that will later be applied to a loaded pandas DataFrame, and the execution result will be sent back as an answer, which is precisely the approach used in the *Chat2VIS* [96] method. To ensure robust performance and experiment with different agent setups, additional details and conditions were incorporated into the project. First, the concept of **LLM Experts** was introduced: one or several specialized models are called with different prompts to solve a smaller task. Expert LLMs in this project can be categorized into three types.

The **Planner LLM** draws inspiration from the Plan-and-Solve (PS) method [83], where the LLM is tasked with breaking down the given task into several subtasks designed to simplify the subsequent code generation process. The **Coder LLM** is prompted to generate Python code to answer the user's question. The input prompt may be augmented with subtasks identified by the Planner, theoretically reducing the likelihood of producing incorrect or irrelevant code. The **Debugger LLM** expert is activated in a loop whenever the code generated by the Coder leads to an exception or fails to produce the desired output. This debugging loop continues until the code executes successfully or a predefined maximum number of iterations is reached. The input prompt for the Debugger can include the initial user query, the previously generated code, and the exception message.

Moreover, the experimental section of this thesis introduces different evaluation approaches, leading to the development of two distinct code generation styles: *"simple"* and *"functions"*. The *simple* style involves directing a Coder LLM to generate a code snippet that executes within a 'main' block, without the need to create a separate function to fulfill the user's request. The text answer is simply retrieved from `print()` statements, generated directly by an LLM. In contrast, the *functions* style involves prompting an LLM to either generate or fill in a specific function (i.e., `def solve(df: DataFrame)`) that can later be called on a DataFrame object, allowing the execution result to be conveniently captured in a variable. This method facilitates direct comparison with a reference output, ensuring accurate and straightforward 1-to-1 evaluation.


User Query:
What weather prevailed during June?
Tag Query Type
Planner LLM
GPT or any other general reasoning model
Coder LLM
GPT, CodeLlama, WizardCoder, Magicoder
Debugger LLM
Any capable of debugging LLM
Filtering Formatting Executing
Possibly showing or saving a chart depending on the tagged query type
Output:
Sunny

PragueWeather23

T (C)	Type	Date
22.8	Sunny	6.8.
13.37	Rainy	7.3

1, Identify rows date in June
2. Aggregate the Type
3. Sort the aggregated df
4. Return the first row

```
import pandas as pd

def solve(df):
    ...
    return weather
```

Figure 3.2: Flow diagram of the implemented LLM-based agent system.

The proposed scaffolding program, also referred to as an Agent, is schematically depicted in Fig. 3.2. The program follows these sequential steps to respond to the user's query:

1. The program receives the user's query and the selected table.
2. The query is automatically classified/tagged into one of two categories: *"plot"* for visualization intents or *"general"* for textual TableQA queries.
3. Optional step: The Planner LLM is prompted to outline a sequence of steps to address the query without directly generating code.
4. The query, potentially augmented with a plan, is then used to create a prompt through a selected strategy. The prompt strategies are described

in details in one of the subsequent sections.

5. The Coder LLM generates Python code which utilizes standard libraries such as Pandas, Matplotlib, and Numpy.
6. The generated code is filtered, formatted, and executed. Depending on the results, the Debugger LLM may be invoked. The output is either returned as a text response or, in the case of an image, stored in a folder or displayed interactively.

The following subsections provide detailed descriptions of each of the program modules.

3.2.1 Agent module

Agent module contains the creation and high-level operation of the Agent itself. It supports flexible configuration options, allowing for customization of the language models used, the level of detail in data descriptions, and strategies for debugging and code generation. The *answer_query()* method is the main communication point. It controls the flow of the agent, handling condition branching, query tagging, filenames, etc. It calls the LLMs to plan, write code and debug sequentially. To collect valuable data, the method returns a set of information about each step of the process, including execution results, tagged query type, filenames, and prompts used.

3.2.2 LLMs module

This module handles the calls to the LLMs for multiple purposes. First and most importantly, it calls the prompt configuration module and the code generation LLMs depending on the parameters set to the agent. Then it can also call planner LLM or call the OpenAI AssistantsAPI to solve the entire problem. The query type tagging is also performed here.

Tagging

Tagging [21] is the process of labeling text into several, not only predefined, data fields. It was introduced into this project because one of the possible use cases for the agent is to call it inside a back-end application. Therefore, to send data via an endpoint, the agent must first classify the user's intent, whether he wants a text or an image answer. Then, the last line of generated code will either contain a statement to save the image to a storage folder (i.e., $plt.savefig(''filename''))$ to further send it from the endpoint or to display it interactively on the screen for local runs, e.g., in Jupyter Notebooks (i.e., $plt.show()$). In the first case, the filename of the image can be set as an object constructor argument, or it will be generated randomly. After the image file has been sent as a response to the endpoint request, the image

---

[21] `https://python.langchain.com/docs/use_cases/tagging/`

could be safely automatically deleted from the machine.

The default tagging strategy that was implemented for the project was using OpenAIs functions. In particular, tagging is performed with a help of a Langchain pipeline that takes in a Pydantic[22] class definition that is then parsed to the format of an OpenAI function and processed by an LLM, see Listing 1. This high-level approach is an innovative way to define several LLM call templates in a visually appealing way.

```
from langchain.pydantic_v1 import BaseModel, Field

class Tagging(BaseModel):
    """
    Tag a piece of text with specific information.
    Classify if the user requested a visualization,
    e.g., a plot or graph, or some general numerical result,
    e.g., finding a correlation or maximum value.
    """
    topic: str = Field(description="The topic of the \
            user's query, must be 'plot' or 'general'.")
```

Listing 1: Pydantic class definition. An instance of it is then processed by Langchain and OpenAI API to tag a user query.

To keep the ability of the implemented Agent to run locally without calling external APIs another tagging technique was introduced. DeBERTa[23] model modification was used for it. It was trained for simple zero-shot classification between given labels and for hypothesis checking. Introducing two classes: *"true"* and *"neutral"* along with a hypothesis: *"A plot, a chart, a visualization, or a graph"*, the model outputs two numbers, which go through a softmax function, resulting in two probabilities. The model itself contains 184 million parameters and is a fine-tuned version of the original DeBERTa from Microsoft Research [104].

3.2.3 Coder LLMs module

Due to the fact that multiple LLMs were tested during this project, this module handles all the configurations and calls to the Coder LLMs. This means it defines the generation parameters, quantizes models, and loads PEFT adapters. All these can again be configured through the Agent constructor arguments. Quantization is performed by passing a *BitsAndBytesConfig* object to the *from_pretrained* method of the transformers library to load the

---

[22] https://docs.pydantic.dev/latest/

[23] https://huggingface.co/MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli

base model.

Another important functionality of this module is final prompt formatting for each LLM separately. That is because every model has it's own format that it was trained on. As it was briefly mentioned in the literature review section, Code Llama - Instruct models expect the input text to be between two special tokens: [INST] and [/INST]. Similarly, another integrated LLM - the Magicoder-S-CL-7B[24], prefers the inputs to use *"@@ Instruction"* and *"@@ Response"* notation in a completion manner, Listing 2.

```
MAGICODER_PROMPT = """You are an exceptionally intelligent
coding assistant that consistently delivers accurate and
reliable responses to user instructions.

@@ Instruction
{instruction}

@@ Response
"""
```

Listing 2: A completion-style input template for the Magicoder-S-CL-7B model. The constructed instruction prompt is inserted in place of `{instruction}`.

The module also handles situations where the model's output includes the input. This requires a separate approach for each model. Taking the Code Llama family as an example, to ensure proper extraction of the generated code from the model's output, one must first remove instruction prompts from the Code Llama - Instruct output. Additionally, it is necessary to smartly replace the "<FILL_ME>" token when utilizing the infilling capabilities of the base Code Llama model, or when using Code Llama - Python to cleverly create a completion prompt format.

The learned LoRA adapters can be loaded by providing a folder path containing the weights. The module structure supports activating 2 PEFT adapters on the base model.

3.2.4 Prompts module

This module is designed to serve two main purposes. The first is to store all the prompt templates for the planner, coder, and debugger LLMs. These prompt templates are essentially strings that contain instructions for the model or wrap the same instruction in a completion or infilling-like manner. The second is to define the functions that insert the information relevant to

---

[24] `https://huggingface.co/ise-uiuc/Magicoder-S-CL-7B`

each individual processed query into those templates. Each query can contain these variable details:

- A **user query** text. A question that the user has written, e.g., *"What is the most frequent fruit mentioned in the table?"*.
- A **table** itself. The table should be parsed into a string in any way (discussed in the literature review, Fig. 2.7), so that the language model has crucial information about the column names and the types of values stored in rows. The current implementation supports three types of information to be inserted into the instruction prompt, as shown in Table 3.1. 1). Converting the head of a pandas **DataFrame** to a string with N rows and passing it directly to the prompt formatter. This is the simplest approach, which has the advantage of user-friendliness - the user perceives this part of the prompt in a tabular format. However, although the models are fully capable of dealing with this format, for a large number of table columns the structure is lost and the model could potentially pay attention to unrelated column names and values. This is why the second parsing type was introduced. 2). Creating a **list of JSON objects**, where each object represents one column and contains the column name and sample values from this column. In addition to maintaining structural integrity, this approach also shows the model the individual column data types more clearly. For example, when a column contains string values in the form of *"202304"*, the pandas DataFrame head approach shows it to the model as *202304*, and the JSON approach as *'202304'*, indicating the correct string type and not an integer. 3). Adding a description of each individual column is also supported in the current implementation and will henceforth be referred to as **column annotation**. This is helpful when dealing with nondescriptive or similar column names, where the Agent could have trouble selecting the right ones. Currently, annotations can only be added by the user by specifying a path to the JSON file with the information, however, there appears to be a good idea to generate such annotations automatically, which could be another step in the Agent program flow (Fig. 2.4).
- Then, when the LLM is tasked with generating code to create a visualization and subsequently store it at a specific path, this path and filename are inserted into the prompt. Additionally, using the Plan-and-Solve flow method, the second generation step requires the previously constructed plan to be passed to the prompt, which is accomplished through several specific *prompt strategy types*.

Prompt strategy types

In this work, the term *prompt strategy* is defined as a collection of prompt templates systematically employed in each specific agent flow. These strategies are characterized by the nature of the generation process—namely, instruction, completion, or infilling. Furthermore, the architecture of the agent, which

Parsing	How the table is represented as a string
df.head(2)	The resut of ‘print(df.head(2))‘ is: Robot ID Distance Traveled Object ... Error Codes 0 1 708.1 Broken cord 1 2 941.4 Sensor fail
JSONs list	Here is also a list of column names along with the first sample values for your convenience (each column is represented as a separate json object within a list): [{’column_name’: ’Robot ID’, ’sample_values’: [1, 2]}, ... , {’column_name’: ’Error Codes’, ’sample_values’: [’Broken cord’, ’...’]}]
Column annotation	{"table_name": "robot_execution_results", "description": "table that describes robot execution on a planning/driving task", "columns":[{ "name": "Distance", "description": "distance covered by robot in a run cycle", "type": "float64" }, ...

Table 3.1: Three non-exclusive types of table parsing available in the current implementation. Can be used separately or together in a single prompt.

may or may not include components such as a debugger or planner, as well as the style of code generation—ranging from a straightforward ’main’ script to a function definition—also define the characteristics of the prompt strategy employed. As for now, there are 6 prompt strategy sets and 2 debugger specific prompt strategies, the most distinct and important examples of which that show the nature of each prompt set are attached in the Appendix.

- **Simple** prompt strategy (Listings 7 and 8 examples). Characterized by the Plan-and-Solve flow in the second step of which the code is generated in a form of a simple Python script. This way, the coder prompt contains also the plan generated in the first step. Textual response is then produced by the LLM itself that generates a `print` statement with a result. Saving the created visualization and showing it interactively are placed into two distinct prompts.

- **Coder-only simple** strategy removes the planning step from the Simple strategy and the Coder LLM is asked to generate the Python script without a plan.

- **Functions** strategy asks the Coder LLM to generate a function definition `def solve(df: pd.DataFrame):` that returns values that are supposed to answer the user’s question. For instance, for the question *"By how much the average temperature in spring is larger than autumn?"* the LLM generates a function that returns a single float value. Moreover, the LLM is instructed not to include the `print` statements, so that the consequent result parsing was correct. Such approach enables to evaluate the Agent quantitatively, which is described in the experiments section. Conversely, while generating a block of code with a `print` statement in the end allows for less robust evaluation with another LLM, keyword finding or manual human comparisons.

- **Coder-only functions** follows the same way as the Coder-only simple strategy, removing the planning step. See Listing 9 for an example.

- **Coder-only completion functions** is a strategy that is needed specifically for completion based models, such as Code Llama - Python. Here, the Coder LLM is given the function's signature with a docstring and the model proceeds to generate the body of the function. The docstring then contains the description what that function does and states that the *df* in the arguments is always the same and fixed. Listing 10 for an example of a completion prompt.

- **Coder-only infilling functions**. The core idea is the same as with the completion strategy, however, other than a function signature with a docstring, it also specifies the last line of the body. For the *general*, i.e., textual answers it is a *return result* statement. For plot saving and showing those are *plt.savefig("filename")* and *plt.show()* statements respectively. A special infilling token is inserted in between the docstring and the last line.

- Two debugger strategy sets both consist of a single prompt. **Basic debug prompt** asks an LLM to look at the previously generated code and read the error message, and then write a new code snippet that fixes the error (Listing 11). **Completion debug prompt** is intended for completion models.

3.2.5 Code cleaning and execution

Another crucial step in the scaffolding program is filtering and extracting the executable code snippet from the LLM response. This is handled in the code manipulation module. By default, the LLM is asked to return code enclosed in backticks, with a *python* keyword. However, when working with small local models, it is usually the case that the output will be in another format or it will contain additional information. The model could start to explain its thought process, add descriptions of the used libraries, define a new DataFrame object, or test generated code with subsequent test cases. Moreover, base models that weren't aligned for instructions or fine-tuned (e.g., Code Llama and Code Llama - Python) are not used to stop the autoregressive generation at any time, given the close to zero temperature and no repetition penalties. Those models generate the output until they reach a set max token limit. This is why it is also essential to cut the generated code in certain places so that the code includes only the needed functionalities. To fix possible indentation issues and transform the code into a conventional format, the *autopep8* library is used. All of these cases are considered in this module.

The code execution itself is performed through Python's:

```
exec(code, {'df': df})
```

where `code` is a string containing the extracted snippet and the dictionary defines variables to be pre-defined in the `exec` scope.

For safety purposes, a "blacklist" of keywords was introduced. It is a simple way to ensure that a harmful prompt injection won't compromise the local machine. These keywords include operating system processes, multi-threading, `exec` and `ast.literal_eval` statements, and similar commands.

Chapter 4

Experiments and Results

The main goal of this project is to propose, implement, and compare LLM-based Agents in the task of natural language-controlled tabular data analysis (TableQA). This chapter focuses on the experimentation and observation of how the implemented Agent performs when supported by different LLMs and defined by various parameters. Given that this task has not been extensively researched, this work attempts to discover what works best and how modern LLMs can operate within an agent-like structure. The main research questions that I have sought to answer with the experiment results include:

1. To what extent do smaller open-source LLMs trail behind API-based giants in quantitative measures during the TableQA task?
2. Is it possible to perform task-specific fine-tuning of a small open-source model to achieve performance on par with other state-of-the-art (SOTA) coding LLMs in the TableQA task (compare open-source and API-based models)?
3. What is the overall performance of a generic LLM-based Agent and what are the challenges the system struggles with the most?

To answer these questions, this chapter was divided into several sections: collection and creation of training data for the consequent fine-tuning, the actual fine-tuning setup, proposal and creation of an evaluation benchmark, and finally, the comparison of the Agents and LLMs themselves.

4.1 Training data

The target for training is the Coder LLM. It was decided to stick with the *Coder-only functions* prompt strategy for instruction training and *Coder-only completion functions* for completion training to later compare the performance versus Plan-and-Solve *Functions* prompt strategy. To fine-tune a model this specific task, the training corpora for the Coder LLM must contain instances with a textual input and a desired textual output. However, such dataset isn't trivial to construct, as each input prompt includes information taken from: a tabular file (e.g. *csv*, *xlsx*), a user query, all additional parameters to the prompt (number of rows to sample, column description, etc.). This way, it is needed to:

1. Stick to one prompt parameter configuration, collect a diverse set of table files, and come up with user questions on these specific table files,
2. Construct prompts from each distinctive table-query pair as they would be constructed for the LLM in the Agent pipeline,
3. Collect the desired outputs of the model, i.e., functional Python code in the exact format that is implied by the used prompt strategy.

First, I created a training corpora containing 250 instances manually. A total of 66 tables was collected either by synthetic generation using ChatGPT, where the names of the columns and the general instructions about values were given to the model to produce a *.xlsx* file, or by collecting and slightly modifying the tables from sources such as WikiTableQuestions[1]. The majority of 250 questions that query these table files were designed to align with the general tasks that are needed to be accomplished by the trained system with around 50 of those being taken and modified from WikiTableQuestions, which also provides the correct answers for each query. A total of 65 questions out of 250 are dedicated to visualizations - the user asks to create a chart of some kind. Each instance has a boolean field *has_plot_answer* that indicates that the correct output has to be an image. This will allow to evaluate the tagging accuracy, where the LLM decides between *general* and *plot* types of prompts. Collected datasets also include fields containing information about prompts used, correct answers to the questions, generated codes, and the filenames of the correct chart images which are stored in the `datasets/` folder of the repository.

Other training instances were extracted from the Text2Analysis dataset. The queries for Forecasing and Basic Insights were left out, because of the first category being outside of the goal of this work and the second providing code with unknown custom functions that usually output simple *"yes"* or *"no"*. Initial fine-tuning experiments also left out the chart generation, as the main focus was to accurately measure the model's performance on *general*, i.e. math and statistics queries. This way, a total of 486 instances were processed and formatted. The code was automatically rewritten for the completion code generation training and the correct outputs in a numeric form were collected by running these new `solve(df)` functions.

This collection process resulted in two JSONL files that contain instances with *input* and *output* fields: one for completion generation style and one for instruction generation. An example of a single training instance is depicted on Listings 3 and 4. This is one of the simpler tasks, but also the one requiring the distinguishing of the date-time column type correctly.

---

[1] `https://ppasupat.github.io/WikiTableQuestions/`

```
'''```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

def solve(df: pd.DataFrame):
    """ Function to solve the user query: 'Revenue with
    Launch Date in January'.

    DataFrame `df` is fixed. The resut of print(df.head(2))
    is:
         Campaign Owner  Campaign Name  Launch Date ...
    0  Halima, Yakubu    Late Jan Email  2023-01-27 ...
    1  Kovaleva, Anna  Billboards small  2023-01-29 ...

    Here is also a list of column names along with the first
    sample values for your convenience ...:
    [{'column_name': 'Campaign Owner', 'sample_values': [...
    ..., 'Engaged Users', 'sample_values': [465, 500]}]

    Args:
        df: pandas DataFrame

    Returns:
        Variable containing the answer to the task 'Revenue
        with Launch Date in January' (typed e.g. float,
        DataFrame, list, string, dict, etc.).
    """
'''
```

Listing 3: Input part of a single completion training instance as a Python string. Ellipses (...) indicate the omission of some words to better fit the text on the page.

```
'''    df0 = df[df['Launch Date'].dt.month == 1]
    revenue_with_launch_date_in_january = df0[["Launch Date",
                                            "Revenue"]]
    return revenue_with_launch_date_in_january
'''
```

Listing 4: Output part of a single completion training instance as a Python string. Four spaces at the string start are for the input-output concatenation.

The average input length for the completion style instances is 1807 characters, which is in range of 200 to 450 tokens. The average output length is 378 characters, which is usually below 100 tokens. Mostly, the dataset contains queries on smaller tables that have under 10 columns, which results in a shorter input length.

Campaign Owner	Campaign Name	Launch Date	Revenue
Halima, Yakubu	Late Jan Email	27.I	$6 980
Kovaleva, Anna	Billboards small	29.I	$4 732
Smith, Avery	Billboards large	03.II	$5 632
Lawson, Andre	Product review 3x	16.I	$5 676
Cartier, Christian	Targeted - Group 1	26.I	$136
	Billboards small	03.I	$8 703
Barden, Malik	Industry Conference	23.II	$4 540

Table 4.1: An example of a table that is queried in one of training instances. Four out of eight columns are displayed for convenience.

The Code Llama family of language models was selected as a target for fine-tuning experiments. Due to its popularity and the simplicity of its architecture, it seems to be a good foundation upon which to base the research. This study may later serve as a reference for other enthusiasts and researchers. Looking a little further into the evaluation section, Code Llama - Instruct and Code Llama - Python significantly outperformed the base infilling Code Llama. Consequently, it was decided to perform training with instruction-style inputs on the Instruct model and with completion-style prompts on the Python model.

The main focus of the fine-tuning was to enforce correct formatting of the generated code (using backticks) and to promote better reasoning by the model. Correct formatting is crucial for two reasons. First, post-generation code processing and execution require it. Sometimes, the Code Llama models tend to overlook the formatting instructions and generate the output in their own way, often including unnecessary comments, notes, and redundant code. The second reason concerns inference cost and execution time. The most costly aspect of the generation is not the input but the number of output tokens. When prompting base Code Llama models with a task, the redundant explanations, comments, and code testing parts significantly increase the number of output tokens. Conversely, when the LLM has learned to generate concise and straight-to-the-point code that spans only a few lines of Python, the generation time is dramatically reduced. The insertion of comments and self-explanatory variable names is designed to help maintain a linear reasoning process, preventing the model from deviating from the correct solution. By 'reminding' itself of the actual analysis tasks, the model can more clearly and autoregressively focus on the intended tasks.

As part of this research, another dataset containing only Python code using

Pandas was collected[2]. The intention for it was to pre-train a base model on a general corpus without any formatting and using a small learning rate, so that the model had a better idea about all the available functionalities in the Pandas framework. This dataset consists of 3 parts and totals around 5000 instances. First two parts come from filtered DS1000 and OSSInstruct datasets. Filtering was performed by matching the presence of keywords that occur most frequently while using Pandas library, e.g. `DataFrame`, `pandas` or `df`. Then, the third part of this corpus comes from a web-scraped pandas documentation with definitions of all the class names and available methods. Then, for each instance, the `gpt-3.5-turbo-instruct` model was called and asked to generate a usage example along with a comment what this method is supposed to do or which values it returned, Listing 5. Most of the time the examples were already in the web-scraped text, however, the format wasn't consistent and it would worsen the data quality.

~~~
Instance: pandas.plotting.scatter_matrix
Output:
```python
df = pd.DataFrame(np.random.randn(1000, 4),
            columns=['Height','Weight','Age','Income'])
pd.plotting.scatter_matrix(df, alpha=0.2)
```
~~~

Listing 5: A single example out of 2130 Pandas documentation examples. This one shows the usage of `scatter_matrix` method.

4.2 Fine-tuning

This section focuses on enhancing the performance of natural language data analysis, specifically improving answer accuracy and inference speed by utilizing LoRA and QLoRA PEFT techniques to train Code Llama models on the collected training corpora. It describes the training process and highlights relevant details.

4.2.1 Setup

All the experiments and the training process should strive to be reproducible and that's why MLOps techniques were applied. Given that the two datasets are fixed, it is needed to ensure logging and storage of all the parameters used for every run of the training. As described in the Tech Stack section of this work, configuration files and Weights&Biases (W&B) were the main pillars to rely on. Combining two of theses technicalities with setting a fixed random seed for each training cycle we achieve full reproducibility of the training.

---

[2] `https://huggingface.co/datasets/poludmik/pandas_documentation`

The configuration file is passed to the W&B logger at the start of the run and is stored along with the training progress insights. The experiment name is generated using the current time stamp and by clicking a corresponding view on the W&B project page[3] everyone can access those details.

LoRA and QLoRA PEFT techniques were both experimented with. To run QLoRA fine-tuning I modified the official GitHub code to support training with my custom datasets as it required some changes[4]. For LoRA training I used the `peft` and `transformers` libraries from HuggingFace which support seamless training interfaces that require passing all of the hyper-parameters to the `Trainer` object and running the `trainer.train()`.

The output of each training method is a model adapter with learned weights. The best performing adapters, as well as the Pandas pre-trained one, are stored on my HuggingFace profile[5].

There are several hyper-parameters that were experimented with during this work. Let's briefly look at the most significant ones:

- **Learning rate.** There are two things to set. First, the learning rate scheduler type (a function), and secondly, the values for the function initialization. I've experimented mainly with two scheduler types: *cosine* and *constant*. The cosine scheduler for learning rates is valuable in machine learning for its smooth, gradual reductions that help prevent training instability. It effectively balances higher early learning rates for navigating noisy gradients and lower rates later on for precise convergence, leading to improved generalization on test data. Combined with a warmup it is often a well-suited scheduler for general deep learning training. The values for the learning rate were under 0.0005, but varied across the experiments.

- **LoRA hyperparameters**. Target modules are a set of specified components within the model's architecture where low-rank matrices will be applied to modify the existing weights. Given the following set of weight layers some of them were selected for every training run:

  $$gate\_proj, q\_proj, v\_proj, k\_proj, o\_proj, down\_proj, up\_proj$$

  All of which refer to the components of the transformer architechture, i.e. q, k, and v projections refer to the Queries, Keys, and Values in the attention mechanism. r and α parameters set the dimensions and the impact of the new weights. Mostly, those were kept low with $r \in \langle 8, 32\rangle$ and $\alpha \in \langle 16, 64\rangle$. For instance, when conducting LoRA fine-tuning on Code Llama 7B model on all 7 target modules with $r = 8$ and $\alpha = 16$, the total number of trainable parameters is $19,988,480$. Relatively to

---

[3] `https://wandb.ai/poludmik/codellama_LoRA?nw=nwuserpoludmik`

[4] `https://github.com/poludmik/qlora_for_codegen`

[5] `https://huggingface.co/poludmik`

the total number of model parameters $(6,758,404,096)$ it takes up only 0.296%.

- **Evaluation** was performed every 5 steps on validation data. The validation data was $\approx 2\% \pm 1\%$ of the training data.
- Total number of **steps** usually varied depending on the number of training instances, mostly corresponding to 1.2 epochs. And the learned weights were saved every 25 to 50 steps. The **optimizer** for LoRA was always set to `paged_adamw_32bit`.

4.2.2 Comparative Insights and Observations

First, let's briefly look at how the fine-tuning run looked for the general Pandas corpus pre-training, Fig. 4.1. Two runs were conducted: on all 3 corpora (including filtered OSSInstruct and DS1000) and solely on Pandas documentation examples. The learning rates were set really low together with constant schedulers. The intention was to slightly refresh the base model's memory on Pandas usage. The evaluation loss doesn't decrease to 0, which means that the model+adapter combination did not over-fit on the training data. Training losses follow the same pattern as the training was performed in under one epoch and the model did not even see a new training instance that it was provided with. The only target module was *gate_proj*.

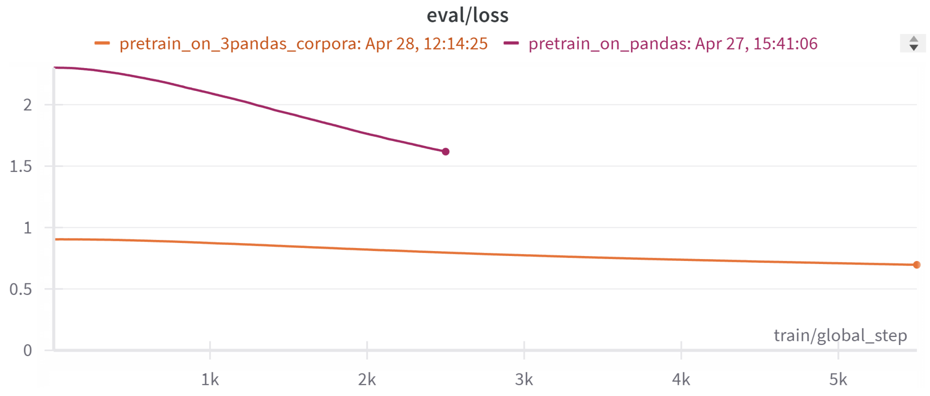


Figure 4.1: Smoothed out evaluation losses during adapter training on 3 pandas corpora and pandas documentation corpus.

When looking at several training runs on the completion task for the Code Llama 7B-Python model, the first thing to notice is how similar the losses appear. Due to the fixed random seed, the training instances are processed in the same order each time, and the only variable is the hyperparameters, which affect the gradient values and hence the speed of convergence. Fig. 4.2 depicts this pattern. The interactive report is also publicly available online[6]. By examining the hyperparameters, it is observed that LoRA adapters focusing on

---

[6] `https://api.wandb.ai/links/poludmik/inz1dpvb`

more target modules tend to converge faster. This underscores the importance of selecting the correct learning rate, which allows for at least one epoch of training where each instance contributes to near-optimal gradient updates.

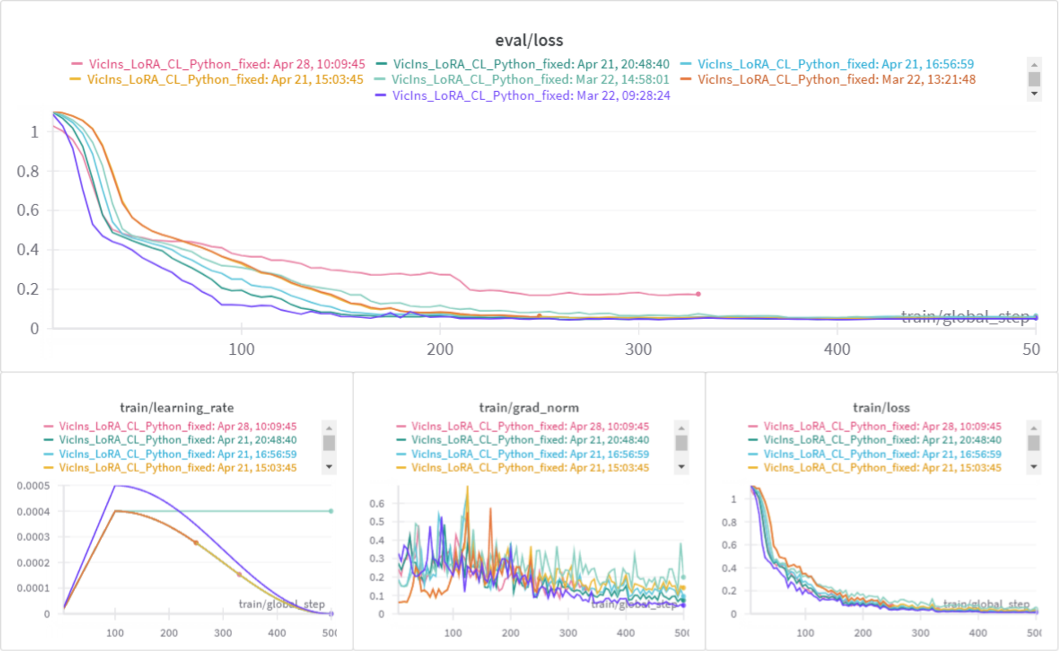


Figure 4.2: Comparison of CL-7B-Python for completion training cycles. Random seed stays the same, hyperparameters are different for every run.

The only exception to the loss pattern is the upper pink line, which was trained on a modified and slightly expanded training corpus. Here, 30 additional training instances were added, thereby shuffling the order of instance processing.

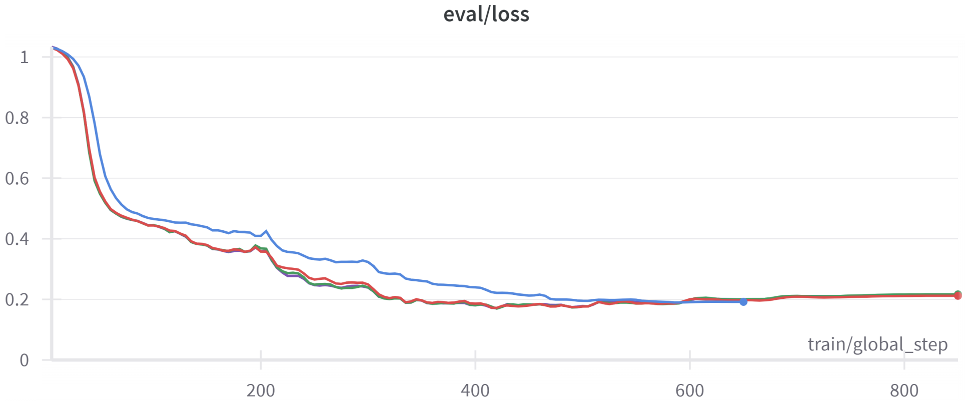


Figure 4.3: Training second LoRA adapter on top of firstly pre-trained on Pandas corpus adapter.

When examining the validation losses of the training runs where the model was fine-tuned atop a pre-trained general Pandas corpus adapter (Fig. 4.3), it is evident that training from scratch takes more time to converge. There is also noticeable slight overfitting on the training data, as indicated by the validation loss beginning to increase after 500 steps. The loss function does not converge to zero because a single evaluation sample encapsulated a long code that returns a dictionary with multiple key-value pairs; the training corpus did not contain similar instances.

```
    ...
    Returns:
        Variable containing the answer to the task 'What is
        the average SoC start' (typed e.g. float, DataFrame,
        list, string, dict, etc.).
    """
    # Your code here
    df['soc_start'] = df['soc_start'].astype(float)
    df['soc_end'] = df['soc_end'].astype(float)
    df['soc_diff'] = df['soc_end'] - df['soc_start']
    df['soc_diff'] = df['soc_diff'].astype(float)
    df['soc_diff'] = df['soc_diff'].replace(np.inf, np.nan)
    df['soc_diff'] = df['soc_diff'].replace(-np.inf, np.nan)
    df['soc_diff'] = df['soc_diff'].replace(np.nan, 0)
    df['soc_diff'] = df['soc_diff'].astype(float)
    df['soc_diff'] = df['soc_diff'].astype(int)
    df['soc_diff'] = df['s
```

~~~
    ...
    Returns:
        Variable containing the answer to the task 'What is
        the average SoC start' (typed e.g. float, DataFrame,
        list, string, dict, etc.).
    """
    # Calculate the average SoC start
    average_soc_start = df['soc_start'].mean()
    return average_soc_start
```</s>
~~~

Figure 4.4: Comparison of outputs for a base CL-7B-Python model (upper) and the same model fine-tuned with LoRA (lower). The completion generation begins after the triple quotation marks ("""), which mark the end of the input prompt.

Figure 4.4 illustrates the improvements that LoRA fine-tuning has made to the quality of the CL-7B-Python model's generation on a simple user query to calculate the average value of a column. The upper illustration highlights a common issue in the base completion generation, where the model repeatedly generates nonsensical lines of code until reaching the maximum token limit.

Consequently, the code cannot be extracted from the output due to the absence of a return statement or backticks, which would indicate the end of the function. This also results in prolonged inference times. In contrast, the lower code snippet demonstrates how the fine-tuned model efficiently writes code, correctly terminating it with backticks and a special token. The forward pass on a single A40 GPU takes about one second.

The main problem of the CL-7B-Instruct model was to obey the given prompt instruction that states which output format must be followed. Due to the model's instruction alignment it has a strong tendency to generate explanations and testing snippets, often splitting the code into several parts. This is fixed by fine-tuning, after which, the model seems to always generate an output in a desired format with backticks.

4.3 Evaluation Metrics

Several evaluation metrics were developed and implemented in this study to quantitatively compare the performance of different LLMs and Agent configurations. This chapter outlines these metrics, explaining how they are used to assess and compare each configuration effectively.

4.3.1 Numeric Accuracy Benchmark

The most important evaluation metric for this work is the numeric accuracy benchmark, which comprises a set of 206 test cases designed to assess the performance of an LLM-based Agent on general math and statistical problems. These questions are based on large tables with approximately 50 columns, focusing on the automotive industry and vehicle tracking to evaluate how the systems perform in a single commercial use case.

A set of comparison functions was implemented, with each test case being evaluated by one or multiple functions. These functions compare the reference object returned by the reference code with the value returned by the agent, depending on the type of the expected reference object. For instance, for the test query *'Select rows where distance values exceed the 100 to 300 range'*, the expected output is a DataFrame that represents a subset of rows from the given table. The comparison functions could test three aspects: whether all values between the reference DataFrame and the output are identical, whether the number of rows is the same, and whether the column names are correct. Then, based on the implementation of the test case, either each correct point contributes to the response accuracy or a single point is added if any criteria are met.

Test Cases are organized into Test Suites, with each Test Suite having the same types of reference values, i.e., comparison functions. However, Test Cases within a single Test Suite can have different object values, but the comparison function remains the same.

This benchmark consists of seven separate question sets:

1. **CT** - Core Tests. These tests comprise simple questions that represent the core functionalities an LLM Agent needs to perform. For example, the questions include finding the minimums, maximums of columns, and performing simple row selections.

2. **AE** - Annotations Exploiting tests. This set is designed to assess the model's ability to deduce column names from unclear user queries or to utilize JSON annotations injected into the prompts.

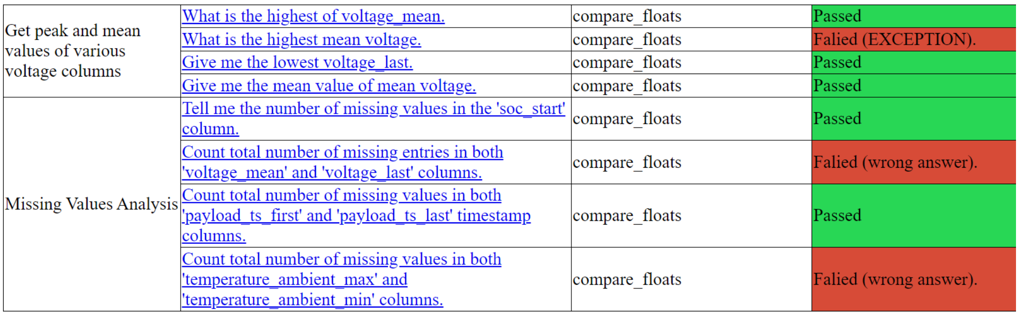
3. **MO** - Math Operations. Primarily focused on testing numeric output values, this set combines simple numerical operations with other DataFrame functionalities performed on a single column. An example is *'What is the minimum of the averaged out maximum speed per grouped car type data?'*

4. **NLU** - Natural Language Understanding tests. The main focus here is to discern the user's intent from unclear queries. For example, to correctly answer the question *'Which trips drained more than half of the battery?'*, the LLM needs to understand what the values in the columns represent, whether they are in percent (50) or floating points (0.5), or something else.

5. **ST** - Selection Tests. These questions focus strictly on filtering a DataFrame and returning either a subset of it or the number of rows or columns. An example question is *"Show rows in a DataFrame where ts_first is not earlier than ts_last."*

6. **CG1** and **CG2** - Comprehensive General tests. These two larger question sets cover all sorts of possible queries. From applying multiple filters, such as *'rows with battery on km bigger or equal to 0.3, current abs diff mean smaller than 19, in str ts first 2023 and timezone is Berlin and the ride was on Wednesday'*, to more complex table modifications like *'Find cycles with high speed variance of more than 5.2 and create a high_variance label.'* CG1 test set also contains some questions that could potentially have multiple right answers. For example when asked *'all vehicle types'* a valid answer could e.g. be a list of strings or a pd.Series object. Each possible reference value is checked and the pass is assigned when at least one of those matches the agent's output.

The output of a benchmark run is an automatically generated HTML report, Fig. 4.5, that comprehensively shows all the details about each test case, generated codes and the resulting accuracy numbers. It also distinguishes agent's mistakes because of an execution exception or a wrong output. In case of an exception, the full traceback of an error is shown next to the generated code.



Get peak and mean values of various voltage columns	What is the highest of voltage_mean.	compare_floats	Passed
	What is the highest mean voltage.	compare_floats	Falied (EXCEPTION).
	Give me the lowest voltage_last.	compare_floats	Passed
	Give me the mean value of mean voltage.	compare_floats	Passed
Missing Values Analysis	Tell me the number of missing values in the 'soc_start' column.	compare_floats	Passed
	Count total number of missing entries in both 'voltage_mean' and 'voltage_last' columns.	compare_floats	Falied (wrong answer).
	Count total number of missing values in both 'payload_ts_first' and 'payload_ts_last' timestamp columns.	compare_floats	Passed
	Count total number of missing values in both 'temperature_ambient_max' and 'temperature_ambient_min' columns.	compare_floats	Falied (wrong answer).

Figure 4.5: An example how each individual test case is highlighted in the resulting HTML report. Two Test Suites are shown and both use the same comparison function to compare to floating point numbers.

4.3.2 LLM as evaluator

Given the reference answer and the textual output from an LLM Agent, it is possible to compare the two with the assistance of a modern and capable LLM. Contemporary models, such as the latest GPT versions, have the ability to determine whether the agent's response aligns with the reference. This metric iterates over all Test Cases from the numerical test described in the previous section, and prompts the `gpt-3.5-turbo-1106` model to assign a score of 1 if the answer is correct, and 0 otherwise. Prompting is conducted by providing the model with several examples of answers that are considered correct according to the ground truth, thereby enhancing the model's ability to generalize across all answer pairs, as shown in Listing 6.

```
- Correct answer: "[2023-03-11, 2023-03-31, 2023-02-19]",
agent's answer: "0   2023-02-19
1   2023-03-11
8   2023-03-31
Name: Last restock date, dtype: datetime64[ns]
", score: 1.

- Correct answer: "CGATCGCCGT", agent's answer: "TGCTTACGGA",
score: 0.

- Correct answer: "0.85", agent's answer: "0.8494521",
score: 1.
```

Listing 6: Few-shot prompting technique. Three out of six examples that are shown to the GPT evaluator.

4.3.3 Tagging Accuracy

It is also crucial to assess the accuracy of the query type classification. For the agent to function as a simple server for a web application, it must first determine whether the user wants a textual answer or an image (since images are sent back differently in responses). This metric thus displays the percentages of correctly classified query types. A handcrafted dataset of 250 instances was used for this purpose. The total time taken to classify all 250 instances is also a subject to be inspected.

4.3.4 Visualization Execution Ratio

The simplest baseline for evaluating the created visualizations is to count the number of successful code executions and compute the accuracy. Inspired by the Text2Analysis paper, this metric represents the upper bound for correctly created plots. For this test, a handcrafted dataset of 250 queries and the Text2Analysis dataset were both considered. Of the 250 instances, 64 are labeled as 'plot', and there are 787 instances of plot creation in the Text2Analysis dataset.

4.4 Results

This section sums up the results of the experiments, evaluating different LLM-based Agents with the proposed metrics. Here, the research questions that were put in the beginning of this chapter are answered based on the gathered numbers.

Table 4.2 presents evaluation metrics for various LLM-based agent configurations on the proposed Numeric Accuracy Benchmark. The *LLM Agent* column details the LLMs utilized by the agent along with additional parameters, such as *(A)*, which denotes the inclusion of JSON column annotations in the constructed prompts. To improve readability, details concerning prompt strategies are excluded. The sole model family utilizing the *completion* strategy is Code Llama Python. All models were instructed with the *functions* strategy, requiring the generated code to include the `def solve(df)` function definition for later comparisons of output objects. The Q column specifies whether quantization was applied to the language model during inference and displays the number of quantization bits employed. The D column discloses whether the agent had the chance to debug potential errors in the generated code. Typically, two attempts were permitted, viewed as a reasonable compromise that allows an agent to fix erroneous code. The debugging model was consistently the same as the coding model. Additionally, one segment in the results table illustrates the combined use of two models to replicate the Plan-and-Solve (PS) method. The `gpt-3.5-turbo-1106` model was responsible for generating plans, which were subsequently incorporated into the `CL-7B-Instruct` prompt. Subsequent columns with abbreviated titles reflect the accuracy percentages detailing how the agent performed on each set of questions from the benchmark. The ACC column indicates the overall accuracy across all tests.

			Numeric Accuracy Benchmark % (Base LLMs)							
LLM Agent	**Q**	**D**	**ACC**	**CT**	**AE**	**MO**	**NLU**	**ST**	**CG1**	**CG2**
CL-7B-Istruct	4	-	49.4	77.1	13.9	47.5	29.0	75.8	36.5	41.9
	8	-	48.0	68.8	30.6	47.5	3.2	33.9	8.3	0.0
	-	-	51.0	83.3	22.2	42.5	29.0	74.2	34.6	48.8
CL-7B-Python	4	-	38.5	79.2	27.8	27.5	19.4	41.9	25	37.2
	8	-	32.4	72.9	5.6	25.0	16.1	35.5	26.9	30.2
	-	-	35.3	75.0	2.8	35.0	9.7	43.6	32.7	27.9
CL-7B, Infilling mode	-	-	17.6	25.0	13.9	15.0	29.0	22.6	5.8	14.0
MC-S-CL-7B	-	-	**60.9**	83.3	27.8	65.0	51.6	66.1	51.9	69.8
CL-13B-Instruct	4	-	53.5	89.6	16.7	57.5	29.0	67.7	44.2	48.8
	8	-	59.6	93.8	13.9	55.0	32.3	77.4	48.1	72.1
CL-13B-Python	4	-	50.3	75.0	27.8	42.5	35.5	74.2	30.8	48.8
	8	-	54.2	85.4	33.3	42.5	38.7	72.6	28.9	62.8
CL-7B-Instruct	-	2	56.7	87.5	33.3	57.5	48.4	75.8	32.7	48.8
MC-S-CL-7B	-	2	**65.7**	87.5	25.0	70.0	51.6	82.3	50.0	76.7
gpt-3.5-turbo-1106 + CL-7B-Instruct	4	-	34.0	60.4	8.3	27.5	48.4	53.2	13.5	18.6
	8	-	48.4	64.6	22.2	50.0	45.2	74.2	38.5	27.9
	-	-	53.2	66.7	16.7	57.5	48.4	77.4	40.4	48.8
CL-7B-Instruct (A)	-	-	**59.6**	85.4	38.9	62.5	45.2	79.0	44.2	46.5
CL-13B-Instruct (A)	8	-	52.9	83.3	27.8	62.5	35.5	64.5	32.7	51.2
CL-13B-Python (A)	8	-	51.9	62.5	36.1	50.0	35.5	74.2	40.4	48.8
Claude3 Haiku	-	2	74.0	91.7	47.2	72.5	61.3	90.3	65.4	74.4
Claude3 Sonnet	-	2	74.4	85.4	50.0	77.5	54.8	95.2	63.5	76.7
Claude3 Opus	-	2	81.7	97.9	47.2	92.5	71.0	96.8	76.9	74.4
Llama3 - 8B	-	-	55.0	91.3	27.8	56.4	44.1	74.2	48.1	34.1
Llama3 - 70B	-	-	76.0	91.3	52.8	89.7	61.8	79.0	71.1	77.3
gpt-3.5-turbo-1106	-	-	78.2	95.8	41.7	80.0	74.2	77.4	78.9	90.7
gpt-3.5-turbo-1106	-	2	78.5	95.8	63.9	75.0	61.3	75.8	80.8	88.4
gpt-3.5-turbo-1106 (A)	-	2	74.4	95.8	44.4	70.0	61.3	82.3	65.4	88.4
gpt-4-turbo-2024-04-09	-	2	86.2	93.8	72.2	90.0	74.2	93.6	88.5	81.4
gpt-4-turbo-2024-... (A)	-	2	84.0	89.6	72.2	92.5	64.5	95.2	86.5	74.4
gpt-4o	-	2	**88.1**	97.9	61.1	90.0	71.0	96.8	88.5	97.7

Table 4.2: Evaluation of agents with base LLMs. **Q** column indicates quantization bits of a loaded LLM ('-' means full precision) and **D** tells how many debugging attempts the agent had ('-' means no debug possibility). In case of Plan-and-Solve combination of GPT + CL the quantization bits are shown for the CL. Symbol *(A)* means the annotations were added to the prompts.

As observed, the 7B Code Llama models consistently achieve up to 51% accuracy. Among these, the `CL-7B-Instruct` model significantly outperforms the Python and Base models, which were prompted for completion and infilling, respectively. The primary challenge for the `CL-7B-Python` model appears to be in the Annotation Exploiting and Natural Language Understanding tests, where its Instruction counterpart is better. Quantization does not markedly affect accuracy, but it does influence inference speed. Among models with 7B parameters, the recent Magicoder-S-CL-7B (`MC-S-CL-7B`) model is the top performer, reaching 60.9% accuracy without debugging enabled. With 2 debugging attempts, the performance of the `MC-S-CL-7B` model improves by 5%. Similarly, providing debugging attempts to the `CL-7B-Instruct` agent yields comparable improvements, showcasing the robust debugging capabilities of these LLMs. Interestingly, adding column annotations to the prompts of the `CL-7B-Instruct` model enhances accuracy more than employing debugging.

Models with 13 billion parameters, quantized at 4 or 8 bits, outperform the 7B Code Llamas. Yet, they still do not surpass the Magicoder-S-CL-7B. Annotations do not enhance the accuracy of the 13B models.

The Plan-and-Solve method offers modest improvements to the quality of code generated by the `CL-7B-Instruct` model, particularly in the Natural Language Understanding test. The observed decline in Core Tests may be attributed to the GPT model generating plans that aim to return more sophisticated values than required. Although the resulting output is technically correct, it is not recognized as such because it exceeds the test requirements. For 4-bit quantized Code Llama models, the Plan-and-Solve method significantly worsens the overall performance.

The API-based Claude3 models exhibit superior performance, significantly outshining the 7B to 13B open-source models. The Haiku variant of Claude3 conducts inferences instantly, delivering results on par with `gpt-3.5-turbo-1106`. Although the Sonnet variant shows only modest improvements over Haiku, the most advanced model, `Claude3 Opus`, exceeds the 81% threshold, showcasing highly sophisticated reasoning capabilities. Achieving such performance necessitates that a model adeptly discern the subtlest nuances and user intentions. It must seamlessly associate user queries with the correct column abbreviations and appropriate return values, which are often implicit.

Similarly, the API-based GPT models generally match the performance of Claude3 models, yet the latest `gpt-4-turbo-2024-04-09` and `gpt-4o` models, with 2 debugging attempts, achieve the highest accuracy at impressive 86.2% and 88.1% respectively. This superior performance is expected as the recent GPT models are designed to function as agents—that is, to write code, initiate execution, and debug if necessary. These numbers higher than Claude3 Opus stem partially from the exceptional performance in the Complex General test sets, which significantly impact the overall accuracy due to their size. Column annotations provide little benefit to GPT models, as these models inherently possess a robust capability to interpret column names effectively.

Answering Research Question #1: To what extent do smaller open-source LLMs trail behind API-based giants in quantitative measures during the TableQA task?

The smaller open-source coding LLMs significantly lag behind the large API-based models in the TableQA task. However, by employing strategies such as enabling debugging or adding additional column descriptions, these models perform only about 10% worse than `Claude3 Haiku` or `gpt-3.5-turbo-1106`. Notably, the `MC-S-CL-7B` outperforms base Code Llama models, as it represents a fine-tuned and thus better-optimized version of the Code Llama model.

		Numeric Accuracy Benchmark % (LoRA PEFT)							
W&B Run ID	**Step**	**ACC**	**CT**	**AE**	**MO**	**NLU**	**ST**	**CG1**	**CG2**
QLoRA-test15631219101	200	53.5	87.5	38.9	57.5	35.5	67.7	46.2	25.6
LoRA gate_only Apr21 15:03:45	150	47.4	83.3	27.8	47.5	45.2	56.5	28.9	34.9
	200	**58.3**	91.7	38.9	55.0	54.8	72.6	32.7	53.5
	400	53.9	75.0	44.4	42.5	41.9	74.2	32.7	53.5
LoRA 5targets Apr21 16:56:59	150	50.0	81.3	36.1	52.5	32.3	66.1	32.7	34.9
	200	57.4	81.3	33.3	57.5	54.8	72.6	40.4	51.2
LoRA 7targets Apr21 20:48:40	100	57.4	68.8	44.4	55.0	51.6	71.0	44.2	58.1
	150	54.8	79.2	41.7	35.0	45.2	75.8	42.3	48.8
LoRA pandas Apr 27 20:42:26	550	58.7	85.4	50.0	52.5	51.6	77.4	34.6	48.8
LoRA 7targets Apr28 10:09:45	250	**60.3**	75.0	38.9	67.5	41.9	83.9	44.2	53.5
LoRA CL-Instruct 7targets Apr22 11:30:44	50	52.2	87.5	27.8	37.5	48.4	71.0	26.9	53.5
	75	55.1	79.2	30.6	50.0	45.2	79.0	34.6	51.2
	100	50.3	81.3	25.0	40.0	35.5	77.4	25.0	48.8
	150	36.9	81.3	13.9	20.0	32.3	56.5	13.5	25.6

Table 4.3: Evaluation of agents equipped with QLoRA and LoRA fine-tuned Code Llama 7B models. Coder-only configurations and the most important results were manually selected and shown here.

Table 4.3 displays the Numeric Accuracy Benchmark results for several fine-tuned Code Llama models. The W&B Run ID column includes training run names, accessible on the W&B project page[7]. While not exhaustive, this table highlights the most significant or best-performing runs. Other experiments conducted did not provide additional insights as they showed results that were either consistent with those presented or were not significant enough to alter the conclusions drawn from these highlighted results. The table features a comparison of the top QLoRA result against various LoRA configurations. The terms *'gate_only'*, *'5targets'*, and *'7targets'* specify the number of LoRA target layers used for adapter application. The designation *pandas* indicates that the model was additionally fine-tuned using a

---

[7] `https://wandb.ai/poludmik/codellama_LoRA?nw=nwuserpoludmik`

pre-trained Pandas adapter. The *Step* column indicates the training step at which the trained adapter was saved. All runs presented show the fine-tuning of `CL-7B-Python` on completion with an exception of the last row, where the Instruct model with instruction training data was used.

Several observations can be made regarding the convergence process. Firstly, it is noted that the use of more target layers accelerates the model's convergence to its optimal result. This acceleration is attributable to the increase in trainable parameters that accompanies additional target layers. Secondly, models tend to quickly overfit the training data, underscoring the necessity of continually evaluating performance on unseen data throughout the training process. This phenomenon is clearly illustrated in the last Run row of the table, where performance significantly deteriorates after an additional 25 or 50 training steps.

Furthermore, regardless of how hyperparameters are set, the fine-tuning runs consistently yield results above the 55% mark. This consistency suggests that for a dataset of this size and for the specific requirements of the TableQA task, the quality of the training data is more crucial than the number of LoRA parameters, target layers, learning rate, etc. Adequately setting these parameters will likely yield favorable outcomes. The most effective LoRA adapter for the `CL-7B-Python` that I achieved demonstrates a total accuracy of **60.3%**, marking a significant 25% improvement over the base model. This also represents the best result among all techniques applied to the Code Llama models. Moreover, these weights also match the performance of the SOTA Magicoder-S-CL-7B model when given only one try to generate the code.

Answering Research Question #2: Is it possible to perform task-specific fine-tuning of a small open-source model to achieve performance on par with other state-of-the-art (SOTA) coding LLMs in the TableQA task (compare open-source and API-based models)?

This study demonstrates that task-specific fine-tuning using LoRA on the Code Llama 7B Python model, executed in completion mode significantly enhances its performance in TableQA. A dataset comprising approximately 200 handcrafted and 400 modified Text2Analysis training instances was utilized. Post fine-tuning, the model's performance aligns closely with the Magicoder-S-CL-7B and the 13B Code Llama models, which are recent open-source models. However, achieving a performance level comparable to API-based models, or larger models such as Llama3 70B, appears to necessitate a substantial increase in both the quality and quantity of training data or an expansion in model size.

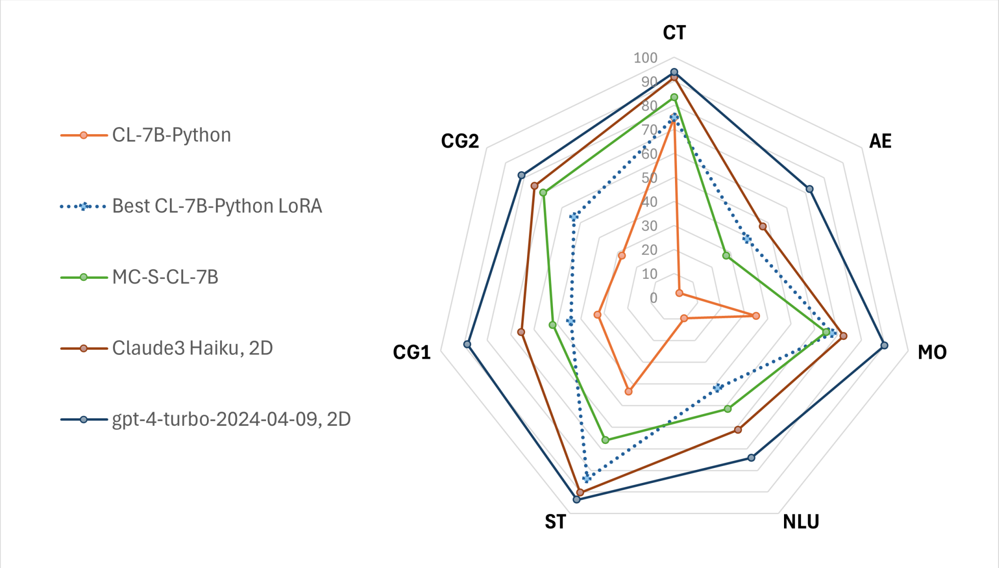


Figure 4.6: Comparison of selected evaluation results showing the improvements of the fine-tuned model against more performant models.

Figure 4.6 graphically illustrates the performance of five selected LLM Agents on the Numeric Evaluation Benchmark. It is evident that the LoRA fine-tuned `CL-7B-Python` model shows substantial improvement across all test sets. The most notable enhancements are observed in the Selection Tests and Math Operations, where its results are nearly equivalent to those of the `Claude3 Haiku`. Such improvements can be attributed to a significant portion of the training data being specifically focused on these tasks. However, in the Natural Language Understanding task, this model still falls short compared to other, instruction-aligned models, due to its inferior reasoning and association abilities, which are typically enhanced by training on a large corpus.

LLM Agent (Coder Only)	LLM-as-Eval Score
LoRA gate_only Apr21 15:03:45 - 200 steps	0.565
LoRA 7targets Apr28 10:09:45 - 250 steps	0.548
gpt-3.5-turbo-1106	0.7
MC-S-CL-7B	0.53

Table 4.4: LLM as evaluator (`gpt-3.5-turbo-1106`) comparing the ground truth answers to best performing agents' answers. The score is the amount of matching answers divided by the number of questions.

Table 4.4 presents the Numeric Evaluation Benchmark accuracies, with scores for each test case assigned by an LLM. The data here strongly correlates with the results obtained when answers are compared programmatically. The models fine-tuned with LoRA achieve even higher scores than Magicoder, further demonstrating the enhancements gained through fine-tuning. The method itself exhibits a degree of conservativeness, with the GPT model cautiously avoiding the assignment of correct labels to incorrect answers. This cautious approach may also stem from the fact that some answers consist of lengthy strings from displayed DataFrames, which pose challenges for the GPT model in terms of comparison.

			Executable Ratio (ER) %	
LLM Agent	**Q**	**D**	**64 from 250**	**787 from T2A**
CL-7B-Instruct	-	-	71.9	78.5
CL-7B-Python	-	-	40.6	63.2
CL-13B-Instruct	8	-	57.8	70.0
CL-13B-Python	8	-	76.6	66.2
MC-S-CL-7B	-	-	75.0	76.2
gpt-3.5-turbo-1106	-	-	85.9	84.6
gpt-3.5-turbo-1106	-	**2**	95.3	90.9
CL-7B-Instruct	-	2	71.9	80.6
CL-13B-Instruct	-	2	68.8	70.1
MC-S-CL-7B	-	2	84.4	80.7

Table 4.5: Ratio of generated codes that were executed successfully for every visualization query from handcrafted 64 and 787 Text2Analysis sets.

Table 4.5 is presented to illustrate the Executable Ratio of visualization queries. It distinguishes between accuracies derived from the handcrafted dataset and those from Text2Analysis. Although this evaluation method provides an upper limit to the true accuracy, it is insightful to explore the effects of enabling debugging and to observe how the `CL-13B-Instruct` model in 8 bits underperforms compared to the 7 billion parameter model. Notably, the Magicoder achieves the same execution ratio as the instruction-aligned Code Llama 7B. Some of the generated plots are in the Appendix C.

Metric	gpt-3.5-turbo	DeBERTa
Overall Accuracy (%)	95.6	82.0
Accuracy on *'plot'* (%)	100.0	86.2
Accuracy on *'general'* (%)	94.1	80.5
Time taken (s)	174.68	178.46

Table 4.6: Tagging accuracy on 250 handcrafted questions. Comparing the usage of Langchain's `ChatOpenAI` Tagging with `gpt-3.5-turbo` to the usage of a small `DeBERTa-v3-base-mnli-fever-anli` classifier model.

The final aspect to assess is the accuracy of query tagging, as shown in Table 4.6. I compare the use of Langchain's `ChatOpenAI` object, augmented with a small Pydantic class that converts to the OpenAI function, with the use of a simple generic classification model, namely DeBERTa. The OpenAI approach achieves 100% success in predicting the *'plot'* class, but it encounters some errors with the *'general'* class. Consequently, it is more likely to generate a visualization for a general text query, which is acceptable because, usually, a visualization can precisely represent the intended answer.

The DeBERTa classifier was introduced to the project as a proof of concept to demonstrate that the designed LLM Agent can operate without internet access. However, its classification performance is inferior, similarly, with the *'general'* class lagging behind the *'plot'* class by 6%. The overall time taken to tag all 250 queries is almost the same for both approaches.

Answering Research Question #3: What is the overall performance of a generic LLM-based Agent and what are the challenges the system struggles with the most?

After conducting several evaluation procedures, it has been observed that an LLM Agent system equipped with state-of-the-art (SOTA) API-based LLMs can consistently perform general tabular analysis tasks without any exceptions in the pipeline. Subjectively, tasks such as Classification, Coding, and Debugging are performed at a level comparable to that of a competent human. The primary challenge for such systems arises from unclear queries, particularly when the user poses a question that is too broad. For systems utilizing smaller, SOTA open-source LLMs, the predominant challenge lies in capturing all intricate details and accurately interpreting the user's intent, which may be vaguely or subtly expressed. For example, selecting the appropriate column names can be problematic if they are difficult to interpret. Also, small open-source LLMs often don't seem to achieve the user request precisely. Generally speaking, these lack maximizing the satisfaction of the user's intended return value with their output. Speaking about code correction, although, these models demonstrate debugging capabilities, there is room for improvement, as smaller open-source coding models are primarily trained on code itself.

4.5 Proposal of Visual Evaluation

The Execution Ratio metric provides a basic assessment of an LLM-based agent's ability to generate visualizations from data. While this metric is straightforward, it does not fully evaluate the plot correctness.

DePlot [105], a model developed by Google, translates images containing data visualizations into textual descriptions, which can then be transformed into structured formats like pandas DataFrames. This capability suggests that visualizations generated by an LLM-based agent could be converted into text and compared against a reference output, which might also be an image or a pre-converted text string.

To explore the effectiveness of DePlot's *plot2text* feature, we[8] tested it on

---

[8] Cooperative work with fellow student Bc. Jan Čuhel. Data aggregation and evaluation.

a collection of 60 visualization images created during the development and testing of the implemented agent program. These images were converted into text and categorized as either *correct*, *weakly correct* (correct in essence but with minor errors), or *incorrect*. The categorization yielded 25 accurately converted images, 14 weakly correct, and 21 incorrect, resulting in 41.67% accuracy for correct conversions and 65% when including weakly correct results.

It's crucial to note that the dataset included not only straightforward bar or line charts but also more complex visuals like confusion matrices, and some unsuccessful plots. These results indicate that the *plot2text* method has an ability to contribute to evaluating the quality of visualizations beyond mere visual checks, providing a quantifiable measure of their accuracy and relevance.

Chapter 5

Conclusion

To summarize, this thesis explores the challenges of analyzing tabular data, discussing its evolution, briefly covering the world of Natural Language Processing (NLP) and Transformer-based Large Language Models (LLMs), their current applications, and the general methods to enhance the quality of Agent-like systems and code generation.

A Natural Language Interface (NLI), an LLM-based Agent program, was developed, allowing for extensive parameter adjustments to alter the agent's configuration. This scaffolding program serves as a robust foundation for future use and modifications.

Datasets specifically designed for training models on TableQA tasks were created and collected, and a comprehensive and multifaceted evaluation benchmark was developed to measure the performance of various LLM Agents.

Furthermore, Code Llama 7B models were fine-tuned in both completion and instruction modes using both LoRA and QLoRA techniques. The process was meticulously documented using MLOps techniques to ensure the experiments are fully reproducible.

Lastly, numerous LLM-based Agents were evaluated and compared using different metrics, highlighting how the fine-tuned models enhanced performance compared to some of the best state-of-the-art open-source coding LLMs, specifically in TableQA tasks. Notably, the performance of the Code Llama 7B Python model improved significantly from 35.3% to 60.3%, matching the performance of the Magicoder-S-CL-7B, one of the top-performing open-source coding LLMs. Major challenges and issues that need to be addressed to further improve the performance of LLM-based Agents in tabular data analysis were also identified and discussed.

Overall, this work provides valuable and broad insights into the use of LLM-based Agent systems for tabular data analysis and aims to serve as a resource for further research, helping others leverage powerful technologies for the benefit of humanity.

5.1 Final Notes and Future Work

In future studies, it would be beneficial to experiment with emerging open-source systems such as OpenCodeInterpreter[107] and QWEN [106], which are coding models known for their enhanced debugging capabilities. Exploring these models could provide valuable insights into improving error detection and correction mechanisms in code generation tasks.

Another area for further investigation emerged from observations made later in this research, regarding the handling of multiple return values in training data. Typically, dictionaries are used for clearer interpretation during training, yet evaluation sometimes involves tuples and other data types. Future work could involve augmenting and improving the existing training dataset with more complex queries that demand extensive reasoning and creating reference codes annotated with comments to clarify the thought process behind the solutions.

Additionally, the integration of DePlot for assessing the quality of plot generation offers a promising direction. By converting visual data into text representations, DePlot would enable a structured evaluation of the outputs produced by LLM-based agents. Integrating this tool would allow to measure the effectiveness of the agents in generating accurate and informative visualizations.

Bibliography

[1] H. H. Goldstine and A. Goldstine, *The Electronic Numerical Integrator and Computer (ENIAC)*, in IEEE Annals of the History of Computing, vol. 18, no. 1, pp. 10-16, Spring 1996, doi: 10.1109/85.476557.

[2] Beyer, H. (1981), Tukey, John W.: Exploratory Data Analysis. Addison-Wesley Publishing Company Reading, Mass. — Menlo Park, Cal., London, Amsterdam, Don Mills, Ontario, Sydney 1977, XVI, 688 S.. Biom. J., 23: 413-414. https://doi.org/10.1002/bimj.4710230408

[3] Vaswani, A. et al. (2017) 'Attention is All you Need,' arXiv (Cornell University), 30, pp. 5998–6008. https://arxiv.org/pdf/1706.03762v5.

[4] Dosovitskiy, A. et al. (2020) 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,' arXiv (Cornell University) [Preprint]. https://arxiv.org/pdf/2010.11929.

[5] He, K. et al. (2021) 'Masked autoencoders are scalable vision learners,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2111.06377.

[6] Arnab, A. et al. (2021) 'ViVIT: a video vision transformer,' 2021 IEEE/CVF International Conference on Computer Vision (ICCV) [Preprint]. https://doi.org/10.1109/iccv48922.2021.00676.

[7] Honda, S., Shi, S. and Ueda, H.R. (2019) 'SMILES Transformer: Pre-trained molecular fingerprint for low data drug discovery,' arXiv (Cornell University) [Preprint]. http://export.arxiv.org/pdf/1911.04738.

[8] Ramesh, A. et al. (2021) 'Zero-Shot Text-to-Image Generation,' arXiv (Cornell University) [Preprint]. http://export.arxiv.org/pdf/2102.12092.

[9] Hong, W. et al. (2022) 'CogVideo: Large-scale pretraining for Text-to-Video Generation via Transformers,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2205.15868.

[10] Sun, Y., Wang, S., Li, Y., Feng, S., Chen, X., Zhang, H., Tian, X., Zhu, D., Tian, H., & Wu, H. (2019, April 19). ERNIE:

Enhanced Representation through Knowledge Integration. arXiv.org. https://arxiv.org/abs/1904.09223

[11] Clark, K. et al. (2020) 'ELECTRA: Pre-training text encoders as discriminators rather than generators,' arXiv (Cornell University) [Preprint]. https://arxiv.org/pdf/2003.10555.pdf.

[12] Ainslie, J. et al. (2023) 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2305.13245.

[13] Su, J. et al. (2021) 'RoFormer: Enhanced Transformer with Rotary Position Embedding,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2104.09864.

[14] Chen, S. et al. (2023) 'Extending context window of large language models via positional interpolation,' arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.2306.15595.

[15] Xiong, R. et al. (2020) 'On Layer Normalization in the Transformer Architecture,' Arxiv [Preprint].

[16] Shazeer, N. (2020) 'GLU variants improve transformer.,' arXiv (Cornell University) [Preprint]. https://arxiv.org/pdf/2002.05202.pdf.

[17] Zhang, B. and Sennrich, R. (2019) 'Root mean square layer normalization,' Neural Information Processing Systems, 32, pp. 12360–12371. https://doi.org/10.5167/uzh-177483.

[18] Devlin, J. et al. (2018) 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,' arXiv (Cornell University) [Preprint]. https://arxiv.org/pdf/1810.04805v2.

[19] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018, April 20). GLUE: a Multi-Task benchmark and analysis platform for natural language understanding. arXiv.org. https://arxiv.org/abs/1804.07461

[20] Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016, June 16). SQUAD: 100,000+ questions for machine comprehension of text. arXiv.org. https://arxiv.org/abs/1606.05250

[21] Zellers, R., Bisk, Y., Schwartz, R., & Choi, Y. (2018, August 16). SWAG: a Large-Scale Adversarial Dataset for grounded commonsense inference. arXiv.org. https://arxiv.org/abs/1808.05326

[22] Yenduri, G., M, R., G, C. S., Y, S., Srivastava, G., Maddikunta, P. K. R., G, D. R., Jhaveri, R. H., B, P., Wang, W., Vasilakos, A., V., & Gadekallu, T. R. (2023, May 11). Generative Pre-trained Transformer: a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. arXiv.org. https://arxiv.org/abs/2305.10435

[23] Ghojogh, Benyamin & Ghodsi, Ali. (2020). Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey. 10.31219/osf.io/m6gcn.

[24] Delétang, G., Ruoss, A., Duquenne, P., Catt, E., Genewein, T., Mattern, C., Grau-Moya, J., Wenliang, L. K., Aitchison, M., Orseau, L., Hutter, M., & Veness, J. (2023, September 19). Language modeling is compression. arXiv.org. https://arxiv.org/abs/2309.10668

[25] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2019, October 23). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv.org. https://arxiv.org/abs/1910.10683

[26] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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. M., Wu, J., Winter, C., . . . Amodei, D. (2020, May 28). Language Models are Few-Shot Learners. arXiv.org. https://arxiv.org/abs/2005.14165

[27] Kaufmann, T., Weng, P., Bengs, V., & Hüllermeier, E. (2023, December 22). A Survey of Reinforcement Learning from Human Feedback. arXiv.org. https://arxiv.org/abs/2312.14925

[28] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023, February 27). LLAMA: Open and Efficient Foundation Language Models. arXiv.org. https://arxiv.org/abs/2302.13971

[29] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., . . . Scialom, T. (2023, July 18). Llama 2: Open foundation and Fine-Tuned chat models. arXiv.org. https://arxiv.org/abs/2307.09288

[30] Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., De Las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M., Stock, P., Subramanian, S., Yang, S., . . . Sayed, W. E. (2024, January 8). Mixtral of experts. arXiv.org. https://arxiv.org/abs/2401.04088

[31] Wu, S., Koo, M., Blum, L., Black, A., Kao, L., Scalzo, F., & Kurtz, I. (2023, August 9). A comparative study of Open-Source large language models, GPT-4 and Claude 2: Multiple-Choice Test taking in nephrology. arXiv.org. https://arxiv.org/abs/2308.04709

[32] Ghosal, D., Chia, Y. K., Majumder, N., & Poria, S. (2023, July 5). Flacuna: Unleashing the Problem Solving Power of Vicuna using FLAN Fine-Tuning. arXiv.org. https://arxiv.org/abs/2307.02053

[33] Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E. (2023, April 21). Evaluating the code quality of AI-Assisted code generation tools: an empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv.org. https://arxiv.org/abs/2304.10778v2

[34] Barke, S., James, M. B., & Polikarpova, N. (2022, June 30). Grounded Copilot: How Programmers Interact with Code-Generating Models. arXiv.org. https://arxiv.org/abs/2206.15000

[35] Chen, M., Tworek, J., Jun, H., Yuan, Q., De Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., . . . Zaremba, W. (2021, July 7). Evaluating large language models trained on code. arXiv.org. https://arxiv.org/abs/2107.03374

[36] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023, May 2). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. arXiv.org. https://arxiv.org/abs/2305.01210

[37] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18). Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[38] Hendrycks, Dan, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song and Jacob Steinhardt. "Measuring Coding Challenge Competence With APPS." ArXiv abs/2105.09938 (2021): n. pag.

[39] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021, August 16). Program Synthesis with Large Language Models. arXiv.org. https://arxiv.org/abs/2108.07732

[40] Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, S. W., Fried, D., Wang, S., & Yu, T. (2022, November 18). DS-1000: A natural and reliable benchmark for data science code Generation. arXiv.org. https://arxiv.org/abs/2211.11501

[41] MULTIPL-E: a scalable and polyglot approach to benchmarking neural code generation. (n.d.). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/10103177

[42] Lachaux, M., Roziere, B., Chanussot, L., & Lample, G. (2020, June 5). Unsupervised translation of programming languages. arXiv.org. https://arxiv.org/abs/2006.03511

[43] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023, May 2). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. arXiv.org. https://arxiv.org/abs/2305.01210

[44] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., & Narasimhan, K. (2023, October 10). SWE-Bench: Can language models resolve Real-World GitHub Issues? arXiv.org. https://arxiv.org/abs/2310.06770

[45] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., & Jiang, D. (2023, June 14). WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv.org. https://arxiv.org/abs/2306.08568

[46] Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., & Jiang, D. (2023, April 24). WizardLM: Empowering large language models to follow complex instructions. arXiv.org. https://arxiv.org/abs/2304.12244

[47] Wei, Y., Wang, Z., Liu, J., Ding, Y., & Zhang, L. (2023, December 4). MagicOder: Source code is all you need. arXiv.org. https://arxiv.org/abs/2312.02120

[48] Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Allie, D. G., Gopi, S., Javaheripi, M., Kauffmann, P., Gustavo, D. R., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., & Li, Y. (2023, June 20). Textbooks are all you need. arXiv.org. https://arxiv.org/abs/2306.11644

[49] Eldan, R., & Li, Y. (2023, May 12). TinyStories: How small can language models be and still speak coherent English? arXiv.org. https://arxiv.org/abs/2305.07759

[50] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., . . . Synnaeve, G. (2023, August 24). Code llama: Open Foundation Models for code. arXiv.org. https://arxiv.org/abs/2308.12950

[51] Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., & Chandra, V. (2023, May 29). LLM-QAT: Data-Free Quantization Aware training for large language models. arXiv.org. https://arxiv.org/abs/2305.17888

[52] Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., & Wei, F. (2024, February 27). The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. arXiv.org. https://arxiv.org/abs/2402.17764

[53] Workshop, B., Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., Gallé, M., Tow, J., Rush, A. M., Biderman, S., Webson, A., Ammanamanchi, P. S., Wang, T.,

Sagot, B., Muennighoff, N., . . . Wolf, T. (2022, November 9). BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv.org. https://arxiv.org/abs/2211.05100

[54] Tian, R., Ye, Y., Qin, Y., Cong, X., Lin, Y., Pan, Y., Wu, Y., Liu, Z., & Sun, M. (2024, January 9). DebugBench: Evaluating debugging capability of large language models. arXiv.org. https://arxiv.org/abs/2401.04621

[55] Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2023, October 3). Large language models cannot Self-Correct reasoning yet. arXiv.org. https://arxiv.org/abs/2310.01798

[56] Chen, X., Lin, M., Schärli, N., & Zhou, D. (2023, April 11). Teaching large language models to Self-Debug. arXiv.org. https://arxiv.org/abs/2304.05128

[57] Zhong, L., Wang, Z., & Shang, J. (2024, February 25). LDB: a large language model debugger via verifying runtime execution step-by-step. arXiv.org. https://arxiv.org/abs/2402.16906

[58] Fang, X., Xu, W., Tan, F. A., Zhang, J., Hu, Z., Qi, Y., Nickleach, S., Socolinsky, D., Sengamedu, S., & Faloutsos, C. (2024, February 27). Large Language Models(LLMs) on Tabular Data: Prediction, Generation, and Understanding – a survey. arXiv.org. https://arxiv.org/abs/2402.17944

[59] Xu, X., Liu, C., & Song, D. (2017, November 13). SQLNet: Generating structured queries from natural language without reinforcement learning. arXiv.org. https://arxiv.org/abs/1711.04436

[60] Zhong, V., Xiong, C., & Socher, R. (2017, August 31). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv.org. https://arxiv.org/abs/1709.00103

[61] Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. (2018, September 24). Spider: a Large-Scale Human-Labeled dataset for complex and Cross-Domain semantic parsing and Text-to-SQL task. arXiv.org. https://arxiv.org/abs/1809.08887

[62] Pasupat, P., & Liang, P. (2015, August 3). Compositional semantic parsing on Semi-Structured Tables. arXiv.org. https://arxiv.org/abs/1508.00305

[63] Y. Luo, N. Tang, G. Li, J. Tang, C. Chai and X. Qin, "Natural Language to Visualization by Neural Machine Translation," in IEEE Transactions on Visualization and Computer Graphics, vol. 28, no. 1, pp. 217-226, Jan. 2022, doi: 10.1109/TVCG.2021.3114848.

[64] Yuanfeng Song, Xuefang Zhao, Raymond Chi-Wing Wong, and Di Jiang. 2022. RGVisNet: A Hybrid Retrieval-Generation Neural Framework Towards Automatic Data Visualization Generation. In Proceedings of the

28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’22). Association for Computing Machinery, New York, NY, USA, 1646–1655. https://doi.org/10.1145/3534678.3539330

[65] Y. Luo, X. Qin, N. Tang and G. Li, "DeepEye: Towards Automatic Data Visualization," 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 101-112, doi: 10.1109/ICDE.2018.00019.

[66] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020, May 22). Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks. arXiv.org. https://arxiv.org/abs/2005.11401

[67] Wang, C., Liu, X., Yue, Y., Tang, X., Zhang, T., Jiayang, C., Yao, Y., Gao, W., Hu, X., Qi, Z., Wang, Y., Yang, L., Wang, J., Xie, X., Zhang, Z., & Zhang, Y. (2023, October 11). Survey on Factuality in Large Language Models: Knowledge, Retrieval and Domain-Specificity. arXiv.org. https://arxiv.org/abs/2310.07521

[68] Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P., Lomeli, M., Hosseini, L., & Jégou, H. (2024, January 16). The Faiss library. arXiv.org. https://arxiv.org/abs/2401.08281

[69] Kathare, Nikita, O. Vinati Reddy, and Vishalakshi Prabhu. "A comprehensive study of Elasticsearch." International Journal of Science and Research (IJSR) (2020).

[70] Zhao, T., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Xu, C., Lin, Y., Zhao, W. X., Wei, Z., & Wen, J. (2024). A survey on large language model based autonomous agents. Frontiers of Computer Science, 18(6). https://doi.org/10.1007/s11704-024-40231-1

[71] Cheng, Y., Zhang, C., Zhang, Z., Meng, X., Hong, S., Li, W., Wang, Z., Wang, Z., Yin, F., Zhao, J., & He, X. (2024, January 7). Exploring Large Language Model based Intelligent Agents: Definitions, Methods, and Prospects. arXiv.org. https://arxiv.org/abs/2401.03428

[72] Li, J., Li, D., Savarese, S., & Hoi, S. (2023, January 30). BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. arXiv.org. https://arxiv.org/abs/2301.12597

[73] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023, February 9). Toolformer: Language models can teach themselves to use tools. arXiv.org. https://arxiv.org/abs/2302.04761

[74] Zhao, T. Z., Wallace, E., Feng, S., Klein, D., & Singh, S. (2021, February 19). Calibrate Before use: Improving Few-Shot performance of language models. arXiv.org. https://arxiv.org/abs/2102.09690

[75] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. ArXiv, abs/2005.14165.

[76] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022, January 28). Chain-of-Thought prompting elicits reasoning in large language models. arXiv.org. https://arxiv.org/abs/2201.11903

[77] Zhang, Z., Yao, Y., Zhang, A., Tang, X., Ma, X., He, Z., Wang, Y., Gerstein, M., Wang, R., Liu, G., & Zhao, H. (2023, November 20). Igniting Language Intelligence: The Hitchhiker's guide from Chain-of-Thought reasoning to language agents. arXiv.org. https://arxiv.org/abs/2311.11797

[78] Wang, X., & Zhou, D. (2024, February 15). Chain-of-Thought reasoning without prompting. arXiv.org. https://arxiv.org/abs/2402.10200

[79] Ye, X., & Durrett, G. (2022, October 31). The unreliability of explanations in few-shot prompting for textual reasoning. OpenReview. https://openreview.net/forum?id=Bct2f8fRd8S

[80] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2022, March 21). Self-Consistency improves chain of thought reasoning in language models. arXiv.org. https://arxiv.org/abs/2203.11171

[81] Zhou, P., Pujara, J., Ren, X., Chen, X., Cheng, H., Le, Q., V., Chi, E. H., Zhou, D., Mishra, S., & Zheng, H. S. (2024, February 6). Self-Discover: large language models Self-Compose reasoning structures. arXiv.org. https://arxiv.org/abs/2402.03620

[82] Zheng, H. S., Mishra, S., Chen, X., Cheng, H., Chi, E. H., Le, Q., V.,& Zhou, D. (2023, October 9). Take a step back: evoking reasoning via abstraction in large language models. arXiv.org. https://arxiv.org/abs/2310.06117

[83] Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K., & Lim, E. (2023, May 6). Plan-and-Solve prompting: Improving Zero-Shot Chain-of-Thought reasoning by large language models. arXiv.org. https://arxiv.org/abs/2305.04091

[84] Drozdov, A., Schärli, N., Akyürek, E., Scales, N., Song, X., Chen, X., Bousquet, O., & Zhou, D. (2022, September 29). Compositional Semantic Parsing with Large Language Models. arXiv.org. https://arxiv.org/abs/2209.15003

[85] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023, May 17). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv.org. https://arxiv.org/abs/2305.10601

[86] Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T. P., Niewiadomski, H., Nyczyk, P., & Hoefler, T. (2024). Graph of Thoughts: Solving Elaborate Problems with Large Language Models. Proceedings of the . . . AAAI Conference on Artificial Intelligence, 38(16), 17682–17690. https://doi.org/10.1609/aaai.v38i16.29720

[87] Do, V., Hoang, V., Nguyen, D., Sabahi, S., Yang, J., Hotta, H., Nguyen, M., & Le, H. (2024, April 3). Automatic prompt selection for large language models. arXiv.org. https://arxiv.org/abs/2404.02717v1

[88] Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q., V., Zhou, D., & Chen, X. (2023, September 7). Large language models as optimizers. arXiv.org. https://arxiv.org/abs/2309.03409

[89] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022, October 6). REACT: Synergizing reasoning and acting in language models. arXiv.org. https://arxiv.org/abs/2210.03629

[90] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023, March 20). Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv.org. https://arxiv.org/abs/2303.11366

[91] Gu, S. (2024, January 12). Mutual Enhancement of Large Language and Reinforcement Learning Models through Bi-Directional Feedback Mechanisms: A Case Study. arXiv.org. https://arxiv.org/abs/2401.06603v1

[92] Li, Y., Zhang, Y., & Sun, L. (2023, October 10). MetaAgents: Simulating interactions of human behaviors for LLM-based task-oriented coordination via collaborative generative agents. arXiv.org. https://arxiv.org/abs/2310.06500

[93] Agashe, S., Fan, Y., Reyna, A., & Wang, X. E. (2023, October 5). LLM-Coordination: Evaluating and analyzing multi-agent coordination abilities in large language models. arXiv.org. https://arxiv.org/abs/2310.03903v2

[94] Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., . . . Tang, J. (2023, August 7). AgentBench: Evaluating LLMs as agents. arXiv.org. https://arxiv.org/abs/2308.03688v2

[95] Sui, Y., Zhou, M., Zhou, M., Han, S., & Zhang, D. (2023, May 22). Table Meets LLM: Can large language models understand structured table Data? A benchmark and Empirical study. arXiv.org. https://arxiv.org/abs/2305.13062

[96] P. Maddigan and T. Susnjak, "Chat2VIS: Generating Data Visualizations via Natural Language Using ChatGPT, Codex and GPT-3 Large Language Models," in IEEE Access, vol. 11, pp. 45181-45193, 2023, doi: 10.1109/ACCESS.2023.3274199.

[97] Fill, Hans-Georg and Fabian Muff. “Visualization in the Era of Artificial Intelligence: Experiments for Creating Structural Visualizations by Prompting Large Language Models.” ArXiv abs/2305.03380 (2023): n. pag.

[98] He, X., Zhou, M., Xu, X., Ma, X., Ding, R., Du, L., Gao, Y., Jia, R., Chen, X., Han, S., Yuan, Z., & Zhang, D. (2023, December 21). Text2Analysis: A Benchmark of Table Question Answering with Advanced Data Analysis and Unclear Queries. arXiv.org. https://arxiv.org/abs/2312.13671

[99] Xu, L., Xie, H., Qin, S. J., Tao, X., & Wang, F. L. (2023, December 19). Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment. arXiv.org. https://arxiv.org/abs/2312.12148

[100] Zhou, Z., Li, L., Chen, X., & Li, A. (2023, July 17). Mini-Giants: “Small” language models and open source Win-Win. arXiv.org. https://arxiv.org/abs/2307.08189

[101] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021, June 17). LORA: Low-Rank adaptation of Large Language Models. arXiv.org. https://arxiv.org/abs/2106.09685

[102] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023, May 23). QLORA: Efficient Finetuning of Quantized LLMS. arXiv.org. https://arxiv.org/abs/2305.14314

[103] Weyssow, M., Zhou, X., Kim, K., Lo, D., & Sahraoui, H.A. (2023). Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. ArXiv, abs/2308.10462.

[104] He, P., Liu, X., Gao, J., & Chen, W. (2020, June 5). DeBERTa: Decoding-enhanced BERT with Disentangled Attention. arXiv.org. https://arxiv.org/abs/2006.03654

[105] Liu, F., Eisenschlos, J. M., Piccinno, F., Krichene, S., Pang, C., Lee, K., Joshi, M., Chen, W., Collier, N., & Altun, Y. (2022, December 20). DePlot: One-shot visual language reasoning by plot-to-table translation. arXiv.org. https://arxiv.org/abs/2212.10505

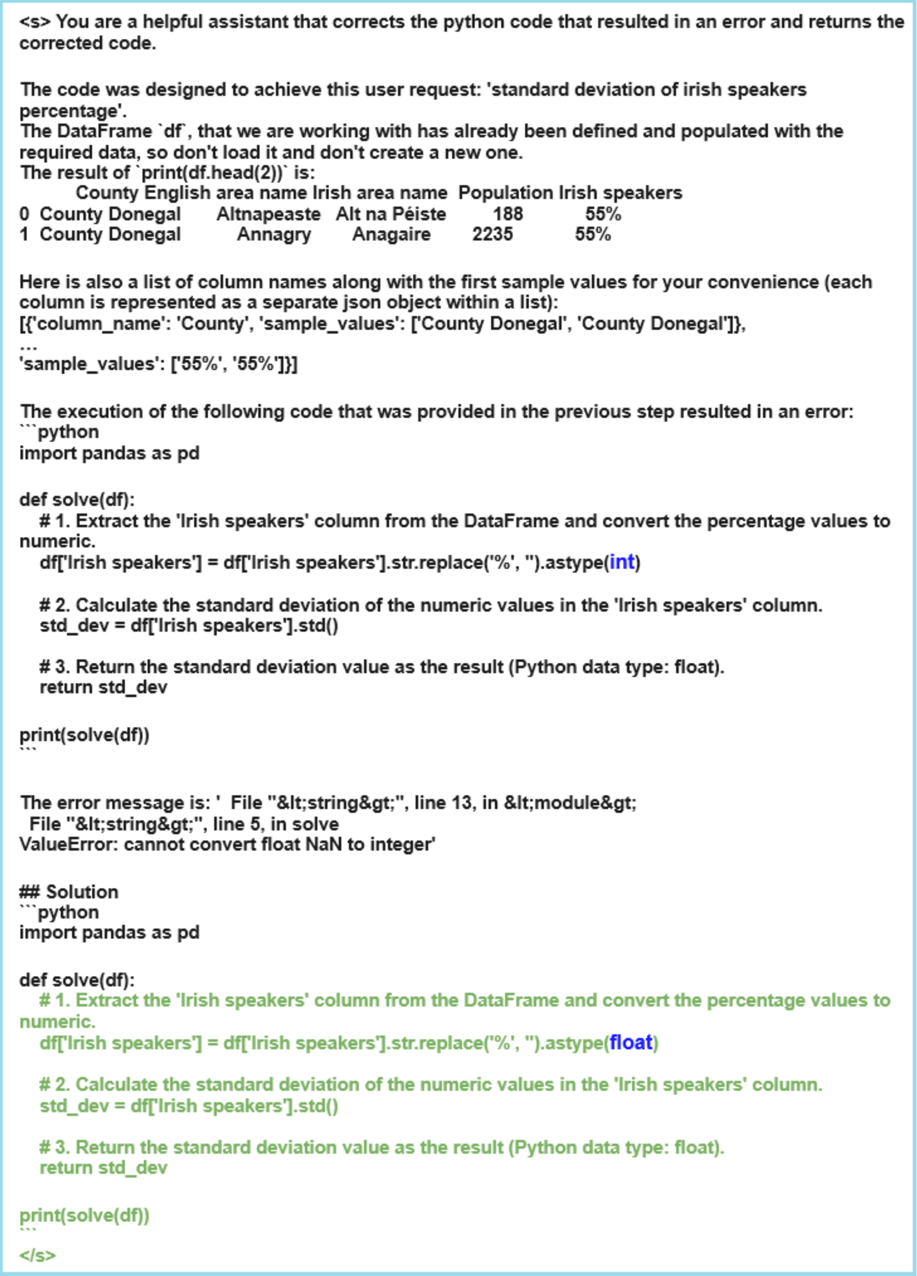
[106] Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., Hui, B., Ji, L., Li, M., Lin, J., Lin, R., Liu, D., Liu, G., Lu, C., Lu, K., . . . Zhu, T. (2023, September 28). Qwen Technical Report. arXiv.org. https://arxiv.org/abs/2309.16609

[107] Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., & Yue, X. (2024, February 22). OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. arXiv.org. https://arxiv.org/abs/2402.14658

Appendix A

Demonstration of basic Code Llama 7B Instruct debugging abilities

Black color is an input prompt, and green color is the output. It successfully changes **int** to **float**, that was implicitly written in the error message.



~~~
<s> You are a helpful assistant that corrects the python code that resulted in an error and returns the
corrected code.

The code was designed to achieve this user request: 'standard deviation of irish speakers
percentage'.
The DataFrame `df`, that we are working with has already been defined and populated with the
required data, so don't load it and don't create a new one.
The result of `print(df.head(2))` is:
         County English area name Irish area name  Population Irish speakers
0  County Donegal      Altnapeaste   Alt na Péiste        188           55%
1  County Donegal          Annagry        Anagaire       2235           55%

Here is also a list of column names along with the first sample values for your convenience (each
column is represented as a separate json object within a list):
[{'column_name': 'County', 'sample_values': ['County Donegal', 'County Donegal']},
...
'sample_values': ['55%', '55%']}]

The execution of the following code that was provided in the previous step resulted in an error:
```python
import pandas as pd

def solve(df):
   # 1. Extract the 'Irish speakers' column from the DataFrame and convert the percentage values to
numeric.
   df['Irish speakers'] = df['Irish speakers'].str.replace('%', '').astype(int)

   # 2. Calculate the standard deviation of the numeric values in the 'Irish speakers' column.
   std_dev = df['Irish speakers'].std()

   # 3. Return the standard deviation value as the result (Python data type: float).
   return std_dev

print(solve(df))
```

The error message is: '  File "&lt;string&gt;", line 13, in &lt;module&gt;
  File "&lt;string&gt;", line 5, in solve
ValueError: cannot convert float NaN to integer'

## Solution
```python
import pandas as pd

def solve(df):
   # 1. Extract the 'Irish speakers' column from the DataFrame and convert the percentage values to
numeric.
   df['Irish speakers'] = df['Irish speakers'].str.replace('%', '').astype(float)

   # 2. Calculate the standard deviation of the numeric values in the 'Irish speakers' column.
   std_dev = df['Irish speakers'].std()

   # 3. Return the standard deviation value as the result (Python data type: float).
   return std_dev

print(solve(df))
```
</s>
~~~

Appendix B

Selected prompt strategy examples

Here, several prompts from the prompt strategy set are shown. All prompt sets can be found in the main Github repository.

```
"""You are an AI data analyst and your job is to assist
the user with simple data analysis.
The user asked the following question: '{input}'.

Formulate your response as an algorithm, breaking the
solution into steps, including any values necessary to
answer the question, such as names of DataFrame columns.

This algorithm will later be used to write Python code
and applied to the existing pandas DataFrame 'df'.
The DataFrame 'df' is already defined and populated with
necessary data. So there is no need to define it again
or load it.
{df_head}
{column_description}
{column_annotation}
Present your algorithm with at most six simple, clear
English steps. Remember to explain steps rather than
to write code. Don't include any visualization steps
like plots or charts. You must output only these steps,
the code generation assistant is going to follow them.

Here's an example of output for your inspiration:
1. Find and store the minimal value in the 'Speed' column.
2. Find and store the maximal value in the 'Voltage' column.
3. Subtract the minimal speed from the maximal voltage.
4. Raise the result to the third power.
5. Print the result.
"""
```

Listing 7: Simple strategy - Planner LLM. Generate a plan for a textual *"general"* answer.

~~~
"""The user provided a query that you need to help
achieve: '{input}'. You also have a list of subtasks
to be accomplished using Python.

You have been presented with a pandas DataFrame named `df`.
The DataFrame `df` has already been defined and populated
with the required data, so don't load it and don't create
a new one.
{df_head}
{column_description}
{column_annotation}
Return only the Python code that accomplishes the following
tasks:
{plan}

Approach each task from the list in isolation, advancing
to the next only upon its successful resolution.
Strictly follow to the prescribed instructions to avoid
oversights and ensure an accurate solution.
Basic ibraries are already imported: pandas as pd,
matplotlib.pyplot as plt, and numpy as np, so you don't
need to import those. You must not include `plt.show()`.
Just save the plot the way it is stated in the tasks.
You must include print statements to output the final result
of your code. You must use the backticks to enclose the code.

Example of the output format with backticks:
```python

```
"""
~~~

Listing 8: Simple strategy - Coder LLM. Create a visualization and save the image.

~~~
"""You are really good with Python and the pandas library.
The user provided a query that you need to help achieve:
'{input}'.

A pandas DataFrame named `df` is fixed. The DataFrame `df`
has already been defined and populated with the required
data, so don't load it and don't create a new one.
{df_head}
{column_description}
{column_annotation}
Return the definition of a Python function called
`def solve(df):` that accomplishes the user query and
returns the result of the analysis (a DataFrame, a list,
a number, a string, etc.).
Basic ibraries are already imported: pandas as pd,
matplotlib.pyplot as plt, and numpy as np, so you don't
need to import those. You must use the backticks to
enclose the code. Do not test the function with anything
similar to `print(solve(df))`, only define the function
to answer the user query, in the format of the following
example:

Example format:
```python
def solve(df):
    # Code to achieve the user query

    # Finally return the result
    return result
```
"""
~~~

Listing 9: Coder-only functions strategy - Coder LLM. Answers the *general*-answer query.

```
'''```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np


def solve(df: pd.DataFrame):
    """ Function to solve the user query: '{input}'.

    DataFrame `df` is fixed.
    {df_head}
    {column_description}
    {column_annotation}

    Args:
        df: pandas DataFrame

    Returns:
        Variable containing the answer to the task '{input}'
        (typed e.g. float, DataFrame, list, string, dict,
        etc.).
    """
    '''
```

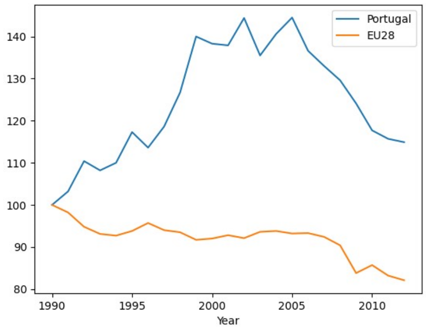
Listing 10: Coder-only completion functions strategy - Coder LLM. Intended to answer the *general* query.

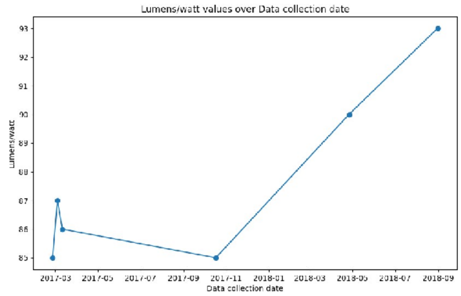
~~~
"""You are a helpful assistant that corrects the Python code
that resulted in an error and returns the corrected code.

The code was designed to achieve this user request: '{input}'.
The DataFrame df that we are working with has already been
defined and populated with the necessary data, so there is
no need to load or create a new one.
{df_head}
{column_description}
{col_annotation}

The execution of the following code that was by a low-quality
assistant resulted in an error:
```python
{code}
```

The error message was: "{error}".

Return only corrected Python code that fixes the error.
Use the same format with backticks.
If part of the code is not defined, or the whole code is a
complete nonsense, define or rewrite it.
"""
~~~

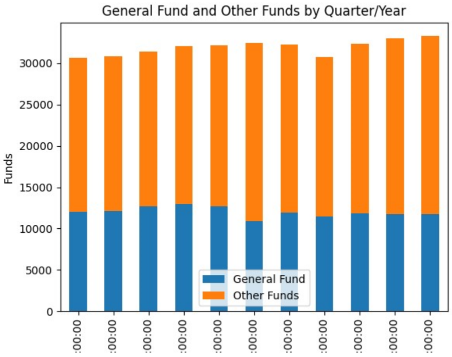
Listing 11: Basic debugging prompt.

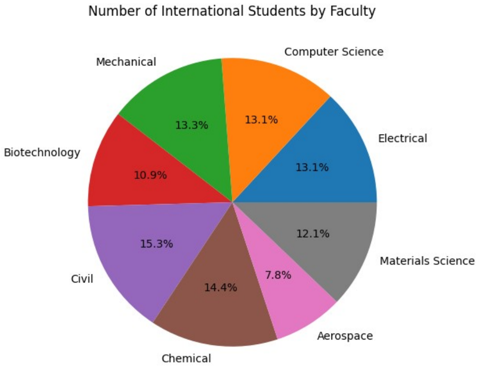
Appendix C

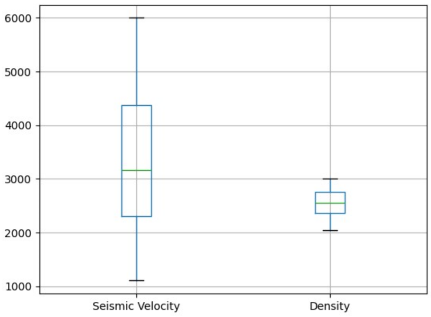
Reference data visualizations produced by LLM Agents

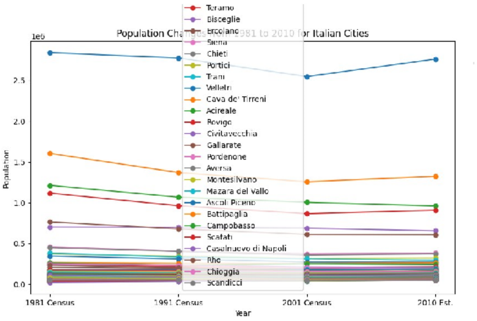

Portugal
EU28
Year


Lumens/watt values over Data collection date
Lumens/watt
Data collection date


General Fund and Other Funds by Quarter/Year
Funds
General Fund
Other Funds
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00
:00:00


Number of International Students by Faculty
Mechanical
Computer Science
Electrical
Biotechnology
Materials Science
Civil
Aerospace
Chemical


Seismic Velocity
Density


Population Changes from 1981 to 2010 for Italian Cities
1e6
Population
Teramo
Bisceglie
Ercolano
Siena
Chieti
Portici
Trani
Velletri
Cava de' Tirreni
Acireale
Rovigo
Civitavecchia
Gallarate
Pordenone
Aversa
Montesilvano
Mazara del Vallo
Ascoli Piceno
Battipaglia
Campobasso
Scafati
Casalnuovo di Napoli
Rho
Chioggia
Scandicci
1981 Census
1991 Census
2001 Census
2010 Est.
Year