UNIVERSITÄT ZU LÜBECK

Using AutoGPT for Information Retrieval Agents

Nutzung von AutoGPT für Informations-Recherche Agenten

Masterarbeit

verfasst am
Institut für Informationssysteme

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Jakob Horbank

ausgegeben und betreut von
Prof. Dr. Ralf Möller

Lübeck, den 15. April 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Jakob Horbank

Zusammenfassung

Abstract

Abstract

Abstract

Contents

1 Introduction 1
1.1 Contributions of this Thesis 1
1.2 Related Work 2
1.3 Structure of this Thesis 3

2 Backgrounds 4
2.1 Information Retrieval 4
2.2 Agents 7
2.3 Language Modeling 8
2.4 Large Language Model Agents 11

3 An Introduction to AutoGPT 14
3.1 The Default Agent 14
3.2 The Forge Agent 16
3.3 The Benchmark System 16
3.4 Project Overview 16
3.5 AutoGPT for Information Retrieval 17

4 Retrieval Augmented Generation Agent 18
4.1 Methods To Improve LLM Generation 18
4.2 Retrieval Augmented Generation 19
4.3 Agent 20

5 Benchmarking for an IR Agent 26
5.1 Existing IR Agent Benchmarks 26
5.2 Custom Benchmarks for Local IR over Journals 26

6 Results 28
6.1 Points of Failure 28
6.2 Benchmarking Results 30
6.3 Subjective Evaluation 30

7 Conclusion 31
7.1 Future Work 31

Bibliography 32

1

Introduction

Encoding knowledge in natural language is an everyday process for humans, but has been a problem for machines for decades.

In recent years, there has been a large increase in language modeling capabilities. After the proposal of the transformer architecture [27], which removed all sequential components from previous language modeling techniques, the architecture and its components were applied to all kinds of language modeling problems. Using massively scaled transformer networks, researchers were able to see emerging behaviors, meaning that the model can solve tasks that it was not trained on.

Humans need to retrieve information from somewhere constantly. It can be a task like digging out an old and hidden memory or finding a book that contains the desired information. Improving the efficiency of tasks like the latter has been extensively researched in the information retrieval area in the last decades. With the introduction of Internet search engines and data repositories, the barrier to accessing information has been drastically improved.

This thesis presents an LLM-based agent that uses retrieval augmented generation to answer user prompts.

The creation of benchmarks for LLM agents is a complex endeavor.

Therefore, I will use a simple question-and-answer dataset to benchmark the agent performance with subjective evaluation.

1.1 Contributions of this Thesis

Three main contributions of this Thesis are

1. An analysis of AutoGPT and its default agent for information retrieval tasks.
2. An agent for information retrieval that uses retrieval augmented generation.
3. A way to benchmark an information retrieval agent using the AutoGPT benchmarking system with a synthetic IR dataset

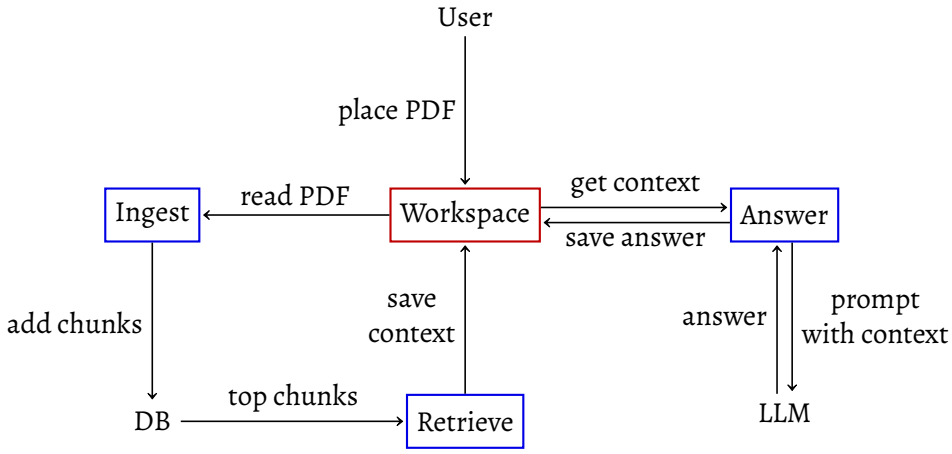

Figure 1.1: The RAG agent manipulates the workspace folder with abilities. To ingest a PDF, the user has to place the file into the workspace. After chunking the PDF, the chunks are ingested into the vector database. With the retrieval ability, the agent can retrieve the top five chunks for a query and save them in a workspace file. The retrieved documents are read from the workspace by the Answer ability. The generated answer is then written back into a file in the workspace.

1.2 Related Work

With the rise of large language models, they have been introduced into different domains. LLM-based agents are a promising way to fulfill user goals autonomously, hinting in the direction of real computer intelligence. Language models have also seen rising popularity in information retrieval applications. Many popular search engines have introduced language models to present the retrieved data to the user. In the following, notable work that is of interest for this work is listed.

LLM Agents

After the possibility of large language models like GPT-3 and then GPT-4 became public, researchers started to test different approaches to LLM-based agents. In particular, the introduction of tools to extend the language model capabilities Toolformer [24] trains a language model to use different APIs in a self-supervised way. For example, the model can use a calculator to externalize math problems, a task category that LLMs struggle with. HuggingGPT [25] uses GPT repeatedly in multiple rounds. For each step, a Huggingface model can be selected by the model to answer the prompt. In a virtual social setting, multiagent communication was demonstrated in [19]. Agents living together in a virtual village have different goals and communicate through natural language.

Building on top of these research experiments with tool usage and access to external knowledge, multiple applications of language model agents have been presented. AutoGPT [26] was the first attempt to make LLM autonomous, but other projects have since been published. An introduction to AutoGPT is given in chapter 3. BabyAGI [17] is a minimal task-driven autonomous agent. With the OpenAI assistants [1] API, users can build

custom assistants that can use models, tools and knowledge to answer user queries.

LLM Information Retrieval

Because of their strong natural language capabilities, many developers work on tools that use these models for information retrieval

In a variety of application contexts, the answers of an assistant have to be correct. This is especially true in research contexts. A model that hallucinates isn't feasible in this case. But while hallucination can be reduced with fine-tuning, it can not be eliminated.

Perplexity AI [20] is an online service that leverages a language model to provide a search that generates an answer from different sources on the internet. The content of the answer is then linked to the found sources, so the user can see and verify the result. As this is a proprietary closed-source product accessible through a web interface, this is not useful to create our custom research assistant. The provided API simply hosts different language models and allows prompting them. There is no possibility of fine-tuning.

1.3 Structure of this Thesis

First, I will introduce the main *backgrounds* that are relevant to this work in chapter 2. The steps of an *information retrieval* system are explained, as well as different strategies to evaluate such systems. Furthermore, I will give an overview of the concept of an *agent* and its main components. Additionally, the *language modeling* section covers the base to understand the current large language models. I will go from the introduction of the transformer architecture to current chat models like ChatGPT. Finally, the combination of language models and agents to *LLM agents* is introduced.

Chapter 3 contains an *introduction to AutoGPT* and its core elements, as well as an analysis of its current information retrieval capabilities. First, an overview of the AutoGPT project and its components is given. I will then explain the default agent in more detail, looking at how the LLM agent ideas are implemented. After gaining an understanding of the AutoGPT project, we will then look at its default capabilities for information retrieval tasks.

In chapter 4, a custom AutoGPT agent that uses retrieval augmented generation for information retrieval is presented. In chapter 5, will deal with how LLM agents can be benchmarking and present a test dataset for the RAG agent from chapter 4. The benchmark uses handcrafted question-and-answer pairs from a scientific humanities journal.

Finally, a conclusion is given in chapter 7. Here I will describe the problems and challenges that LLM agents currently have being a relatively new concept. Additionally, open questions for possible future work are discussed.

2

Backgrounds

Different branches of research were used to build upon this work. These topics and how they are connected to my work are explained in more detail in this chapter.

In section 2.1, I will describe the process of *information retrieval systems*.

The notion of *agents* is described in section 2.2. Agents try to act as rational and goal-oriented decision-makers. In comparison to classical computer algorithms, agents observe, plan and act autonomously in order to complete a task. I will explain the basic concept of an agent and list the most important types of agents. Section 2.3 deals with *language modeling* and the current developments of large language models (LLMs). Building upon the attention-based transformer architecture, modern large language models (LLMs) can handle complex tasks such as text generation. Moreover, the natural language interface of LLMs can be used for tasks in other domains. An example of such a domain is *LLM-based agents*, explained in section 2.4. The text generation ability of the LLM itself is not of interest here, but rather using the internal knowledge of the model to generate a decision of an agent. This section gives an introduction to proposed frameworks and ideas for LLM agents.

2.1 Information Retrieval

Retrieval of information is essential for humans. Information can be anything, like things seen by the eye, thoughts or an article from a book. A broad definition of information retrieval from [15] is:

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Modern information retrieval systems extend this definition with a step dealing with the presentation of the information to the user.

Retrieving the best documents for a query from a large database filled with information of different kinds is a classic problem in computer science. There has been extensive research on database architectures and indexing algorithms. Popular applications of IR algorithms are search engines that enable fast access to billions of documents on the internet. Before retrieving information from an information retrieval system, the data is

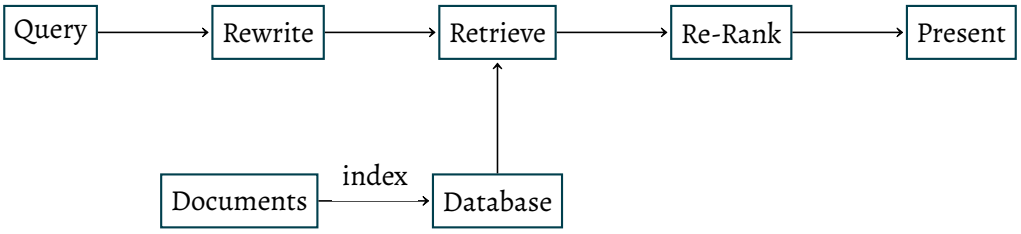

Figure 2.1: Every information retrieval process starts with a user query. The database is then queried to retrieve relevant documents to the query. Before presenting an answer to the user, the retrieved documents are ranked by another relevance measure.

preprocessed and stored. Then a user query initiates a series of steps that can rewrite the query, retrieve relevant documents, re-rank the relevant documents and finally present the information to the user.

Information Retrieval Steps

Information retrieval systems consist of multiple steps. Typically, the user interacts with an information retrieval system by writing out a query that describes the information need. A problem with queries written by humans is that they are not optimized for the following steps of the retrieval system. Human writing sentences can include spelling mistakes missing words or even wrong descriptions of information. In IR systems researchers try to mitigate such problems with query rewriting. A core query rewriting technique is to expand the query with more words to improve the retrieval accuracy.

To retrieve documents that correlate to the search query, different algorithms and models have been proposed. For a long time, functions like the BM25 [22] were often used to retrieve the best matching documents from a corpus. Term-based bag-of-words functions like BM25 rank documents based on word occurrences in every document. With neural network approaches becoming more popular, the focus has shifted toward mapping documents to high-dimensional vectors called embeddings. A similarity matrix can then be used to calculate the distance between to embeddings corresponding to the query and a document.

After a set of relevant documents was retrieved with an efficient retrieval method, the documents are re-ranked by their relevance. This time the algorithms used for ranking the documents are specialized towards quality rather than efficiency. In addition, the re-ranking phase can include task-specific ranking strategies to meet user demands.

In the last step, the information is presented to the user. Large language models have become a popular method to create user-friendly responses from retrieved documents.

Indexing

Before documents can be retrieved from a database, they have to be stored. How documents are stored is a crucial part of creating an information retrieval system. While some data sources are delivered in structured formats like JSON or XML, academic texts are published in journals, articles or conference proceedings. All of these mediums are

distributed as PDFs which is an unstructured data format. The PDF format does not save the content A preprocessing step is required before storing the documents to extract information from unstructured formats. There are different techniques to preprocess text before storing it. With *tokenization*, sentences are split up into words, phrases or symbols. Words with little information such as articles and prepositions are removed from the word list. The words are called *stop words*. Furthermore, a *stemmer* can be used to group words that have the same stem. All of these preprocessing steps help to capture the semantics of a text or sentence.

Evaluation

Information retrieval systems are generally evaluated by categorizing documents as *relevant* or *non-relevant* concerning an information need of the user [15]. The *gold standard* defines which documents are relevant and non-relevant. One has to consider that the relevance decision is made according to an *information need* and not the query. This distinction shows, that a clear information need has to be present in order to effectively evaluate an IR system.

Although information retrieval can be seen as a two-label classification problem, it can not be evaluated with an accuracy measure that is the fraction of correct classifications. As almost all documents will be non-relevant for any given query, the system could achieve high accuracy by simply classifying all documents as non-relevant to the query. For that reason, two measures are used to evaluate IR systems.

Precision is the fraction of retrieved documents that are relevant:

$$\text{Precision} = \frac{\#\text{relevant retrieved}}{\#\text{retrieved}} \tag{2.1}$$

Recall is the fraction of relevant documents that are retrieved:

$$\text{Recall} = \frac{\#\text{relevant retrieved}}{\#\text{relevant}} \tag{2.2}$$

Having these two measures is helpful, as different tasks have different requirements. While high precision might be more important for the average web surfer, a professional researcher might be more interested in high recall. With the two measures trading off against each other, the *F Measure* combines them into a single one. Often written as F_1, the balanced version equally weights precision and recall:

$$F_1 = \frac{2PR}{P + R} \tag{2.3}$$

where P is the precision and R is the recall. These measures are computed on unordered sets of documents. For documents ranked by relevance, the measures are calculated for sets containing the top k documents for different k.

Other properties of IR systems can be measured. The indexing and search speeds are important to enable effective usage of an IR system. Another property is the degree of complexity allowed by the query language. Nonetheless, the ultimate measure is the

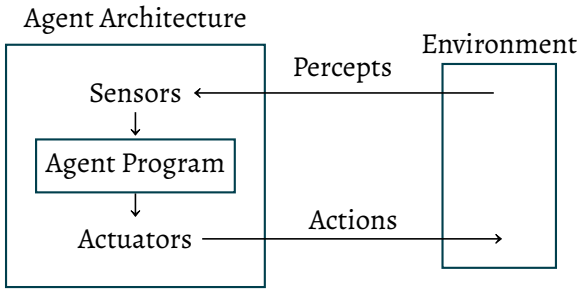

Figure 2.2: An agent as defined in [23]. The agent program runs on the agent architecture. Agents perceive their environment through sensors and act upon it using actuators. The agent program continuously maps perceptions to action and defines the agent type.

user utility. Here, all dimensions of evaluation culminate into a single idea, which makes this very hard to measure objectively. Each user has different tastes and needs for an IR system. The document relevance measurement is therefore a major simplification that can only approximate the user utility of a system.

2.2 Agents

Rationality is viewed as a core component of intelligence. In computer science, computational entities that act rationally are called agents. What it means to act rational is, and will stay an open research question for a long time. However, to use the notion of an agent in practical work a more concrete definition has emerged. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [23]. A human perceives the world through eyes and can act with hands or speech, a software can perceive and act through software interfaces. Both examples can be called agents.

A key step when constructing an agent is to define the *task environment*. The task environment consists of a performance measure, the environment, the actuators and the sensors [23]. Environments can be physical but also purely virtual, we are then using a software agent.

Agent Types

An agent consists of an agent architecture and an agent program. While the agent architecture makes perceptions and actions through sensors and actuators available, the agent program implements the mapping from perceptions to actions. Agent programs can be categorized into four basic types that almost all agents are based on [23]. What kind of agent is useful depends on the given task environment.

Simple reflex agents Simple reflex agents are the simplest kind of agent. They choose actions based on the current perceptions looking at the perception history. This means that even a bit of unobservability can break the agent.

Model-based reflex agents Model-based reflex agents deal with partially observable environments by keeping an internal state of the world. This internal state depends on the perception history. Modeling physical or mental states is a complex topic that is heavily researched.

Goal-based agents In many cases, the perception history is not enough to choose the best action. To be able to do that a goal is required. Goal-based agents search for action sequences that end in a goal state. This process is called planning.

Utility-based agents Even with a goal in mind, there are cases where more than one action sequence leads to the goal state. To decide which action to select, utility-based agents choose the action that maximizes the expected utility. The utility function of the agent should ideally match the performance measure of the task environment.

To improve the performance of an agent it *has to learn*. All agent types can be extended to learning agents. For an agent, learning means modifying each component such that it better aligns with feedback from a new critic component [23]. As a result of these modifications, the agent performance improves.

2.3 Language Modeling

Language modeling is an important research area of computer science. In recent years, developments have significantly sped up with the introduction of deep learning into language modeling. The transformer network [27] has been the base for many advancements in recent years. Deriving from the transformer network, massively scaled-up pre-trained transformer models [5] enabled a big leap in the capabilities of language processing models.

Transformer Networks

Using recurrent structures such as recurrent neural networks and long short-term memory [7] was the dominant strategy in sequence modeling before transformer networks [27]. Every sentence token was represented as a hidden state that is a function of all previous hidden states. While this approach has a reasonable motivation, the sequential nature constrains computation speed for a single training example. This limitation is especially hindering for longer sequences, where batching the training data is only possible to a memory limit. Attempts to minimize sequential computation included convolutional networks that can be computed in parallel [10]. For these models, however, the number of operations required to relate two tokens grows in the distance of their positions in the sequence. Attention mechanisms allow modeling token relationships independent of their distance in a sequence.

The transformer network [27] shown in Figure 2.3 was the first model that removed all recurrent structures and only relies on attention mechanisms. In particular, they use multi-headed self-attention layers. Attention mechanisms learn dependencies between tokens in sentences without regard to their distance. Their non-sequential characteristics allow for better parallelization. Self-attention is an attention mechanism that com-

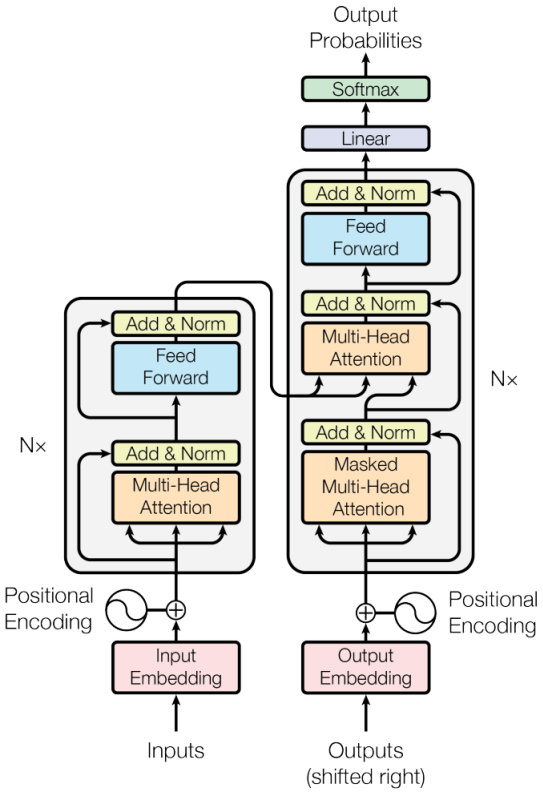

Figure 2.3: The encoder-decoder architecture of a full transformer network [27]. The encoder (left) can attend overall positions to learn rich embeddings. The decoder (right) can generate output sequences. The first decoder attention layer can only attend to previous positions of the input sequence, while the second can attend to the outputs of the encoder. This combines a sequence-to-sequence with autoregressive properties into the decoder.

putes relations between different positions of the same sequence to find a representation of the sequence.

Since the transformer architecture does not use any recurrent structures, the order of the tokens in the sequence must be manually injected. Positional encodings are added to the input embeddings to achieve this.

Like most language modeling networks, a transformer consists of an encoder and a decoder. The encoder has two sub-layers, a multi-head attention block and a fully connected feed-forward block. The decoder is similar to the encoder but has an additional multi-head attention layer that attends to the outputs of the encoder. The first attention layer is masked, and the decoder input sequence is shifted to the right, so only previous positions can be attended by the decoder.

The full architecture has the capabilities of a classic sequence-to-sequence model used for tasks like language translation. All positions in the decoder can attend to all positions in the encoder. For other tasks, however, the individual encoder and decoder sections are a better fit, as explained in the section 2.3 and section 2.3.

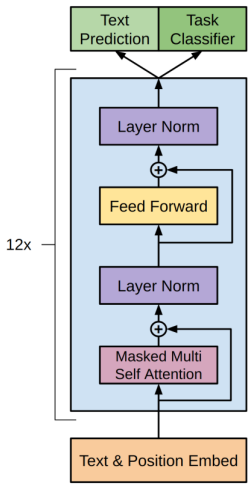

Figure 2.4: The GPT architecture [21]. A transformer encoder without connections to a previous encoder is used. The attention layer can only attend to previous positions in the input sequence. The decoder is stacked 12 times before generating the final output.

Generative Pre-Trained Transformers

Decoder-only transformers are the base of generative pre-trained transformers (GPT). The decoder used for GPT does not rely on the outputs of an encoder, because it is designed for generation tasks. In Figure 2.4 we can see that only the masked multi-head attention layer is kept in comparison to the full transformer network. The simple architecture enables faster pre-training and fine-tuning.

GPT is trained in two phases. In the unsupervised pre-training phase, the model is trained with big datasets of unlabeled text data. The training objective here is to accurately generate the next token of a sequence.

After the first pre-training phase, the model needs to be fine-tuned for a specific task. By including custom tokens in the fine-tuning dataset, the decoder can learn to handle a variety of different tasks such as text generation or classification.

Newer research on generation models focuses on scaling the network. The *scaling law* states that by simply increasing the number of parameters of the network, the model capabilities increase. Furthermore, after a certain level of scaling, abilities emerge that the model was not directly trained on.

As large language models have reached parameter counts of over a billion, only very few companies have the hardware to train or even run inference tasks.

Embedding Models

The left-to-right nature of the transformer encoder suits text generation tasks that only depend on previous tokens. However, there are dependencies between tokens in both directions, which means a decoder-only model can not capture all relations.

The amount of knowledge learned in the pre-training phase is encoded in embeddings. Embeddings are high-dimensional vectors that represent a sequence of words or

tokens. Semantically similar documents are represented with embedding vectors that have a small distance, while semantically different documents have a high distance.

BERT [8] uses the encoder part of a transformer network, to create rich embeddings of input sequences. In pre-training, the bidirectional encoder is trained with two unsupervised tasks. For the first task, random tokens in the sequence are masked out to predict the masked tokens from the remaining tokens in the sequence. In the second task, sentence-level relationships are learned by predicting the next sentence. After pre-training, BERT can be fine-tuned on different downstream tasks leveraging the rich bidirectional embeddings learned in pre-training.

Instruction-Tuned Models

Language models are trained to predict the next token of a sequence. While a lot of knowledge is captured in the weights of the model, most information on the internet is not formatted in a conversational style. Because of this, language models need to be prompted in a specific way to be effective. The prompt needs to be written such that its continuation yields the desired output. This is not optimal for human users, as one would rather write in a conversational style. The training objective of large language models is different from the objective "follow the user's instructions helpfully and safely" [18]. Researchers say that the language model is not *aligned*. A popular attempt at aligning language models is reinforcement learning with human feedback (RLHF) [18].

Instruct models are fine-tuned versions of language models. Capturing the intent of the user is a key challenge for language models. This process is called *alignment*. A popular approach to the alignment problem is reinforcement learning with human feedback (RLHF) [18]. Handcrafted prompts are used to fine-tune GPT-3. The outputs of the model are collected into a set and ranked by humans. This set is then used to train a reward model. With this reward model, the language model is further fine-tuned. The resulting model is called *InstructGPT* and performs better than the baseline GPT-3 model.

Large language models are trained to predict the next token of a sequence, not to follow the instructions of the user. This leads to some unwanted results such as toxic, harmful or fabricated answers that are not true.

2.4 Large Language Model Agents

With the rise of large language models and their impressive capabilities in various language tasks, efforts began to leverage them in agent systems. Experiments to give GPT access to tools [25, 24], that large language models can use external functions if they are prompted with a description of how to access them.

Four key reasons why large language models are suitable as agent brains are outlined in [30]. Agents should act autonomously without direct interventions from humans. The generative capabilities and dynamically adjusted outputs based on the input fit that characteristic.

For practical applications, [33] proposes a distinction between static and dynamic agents. A static agent is characterized as a fixed pipeline that mimics the user behavior.

While this approach works, it cannot deal with complex and sometimes random human actions. The other type of agent can dynamically execute actions that are presented to it. The most prominent way of using LLM for agents is to build a prompt that contains all the information about the task, and then ask it to propose the next action. The prompt can contain descriptions of possible abilities, the task, guidelines, information about the environment and more.

Experiments in [13] showed that large language models focus more on the first and last lines of the prompt in their answer generation. This needs to be respected when constructing long prompts to use LLMs for agents.

LLM Agent Construction

Most large language model agents follow a certain pattern when implementing an agent with LLMs. A construction framework for LLM agents consisting of four modules is proposed in [28]. Their agent framework is composed of a profiling module, a memory module, a planning module and an action module.

Before the actual execution of steps, an agent is given a profile that describes the role of the agent. This profile is usually written into the prompt that is sent to the language model. Profiles can be handcrafted by simply adding a "You are an x" statement to the prompt. Furthermore, profiles can be generated by LLMs which saves time for multiple agents but does not allow for precise control in the profile crafting process.

Information is stored in the memory module. Memory is used to keep a mental state of the environment and store new observations. LLM agent memory types draw inspiration from human short-term and long-term memories. Short-term memory is generally represented by the in-context information given in the language model prompt. It is restricted by the context window length of the transformer architecture. Long-term memory represents an external storage of information. For LLM agents, a popular choice is a vector database, that allows for quick query and retrieval of information.

Planning strategies for agents can be separated into planning with feedback and without feedback. Planning without feedback uses single-path or multi-path reasoning. Techniques like Chain of Thought (CoT) and Zero-shot-CoT prompt LLMs to "think step by step" [29]. This produces a step-by-step solution to a task that can be used as a plan for the agent. Other single-path reasoning approaches like HuggingGPT prompt the language model after each reasoning step [25]. For multi-path reasoning CoT can be used again, now generating multiple plans in a tree-like structure [32]. For complex tasks, the agent should be able to incorporate feedback into the planning process. The feedback can come from the environment, humans or from a feedback model. All of these changes have to be represented in the language model prompt.

To act out the agents' decisions in the environment, the agent needs an action module. There are two main strategies for acting out actions. For the first strategy, the agent uses the task and the memory as prompts to choose the next action in a continuous loop. As the agent proceeds the memory builds up with the action history. This allows for dynamic action choices while the agent is running. The second strategy uses the task to create a fixed action plan that is then executed strictly. Here, the agent can not adapt dynamically. The agent space defines the set of possible actions that can be performed.

Agents can leverage external tools and internal knowledge. Types of external tools can include APIs, databases or other models that specialize in a specific task. The internal knowledge of a language model is also important for the action module. When deciding which action should be used in a step, The planning, conversation and commonsense understanding capabilities are crucial to make a good decision. The outcomes of actions can change the environment, the internal state and also trigger new actions.

For LLM agents we do not have all the components of a classical agent that were listed in the previous section about agent types. [26]

Evaluation

- subjective evaluation - human annotation - turing test - objective evaluation

3

An Introduction to AutoGPT

The general notion of an agent in computer science was introduced in section 2.2. AutoGPT is an open-source project that tries to 'make GPT fully autonomous'. The project quickly gained a lot of traction following its first release. The idea of an LLM that controls an agent sounded like the next step towards real intelligence of computer systems. Lots of programmers joined the AutoGPT Repository, and it grew into a much bigger and now-funded project. Initially, it only contained the AutoGPT agent but over time has seen multiple additions.

The documentation of AutoGPT is sparse, and the project is developed quickly, therefore existing documentation can often be outdated. Therefore, I will give a brief introduction to AutoGPT and its core components in this chapter.

What grabbed the attention of many people is not the whole project, but merely the default agent implementation of AutoGPT. The project further includes *Forge*, an agent framework that can be used to build a custom agent. It defines a minimal agent without logic that abstracts away all the boilerplate. In section 3.2 the Forge agent is introduced and in chapter 4 I use the Forge agent to build a custom agent for information retrieval. While the first iteration of the default agent lived in the terminal, the project now ships with a web application that allows interaction with agents in the browser. Finally, a benchmarking system was introduced. The benchmarking can be used to build level-based test suites, to test the capabilities of an agent. The benchmarking system is introduced in section 3.3

3.1 The Default Agent

The default agent made up the whole project when AutoGPT was first released. The official documentation [11] describes the default agent as:

> A *generalist* agent, meaning it is not designed with a specific task in mind. Instead, it is designed to be able to execute a wide range of tasks across many disciplines, as long as it can be done on a computer.

This generalist design can be seen in the list of available abilities.

Code Execution The agent can execute arbitrary Python code given as a string or a file. The execution is contained in a docker container that can access the agent workspace.

File Operations To access files in the agent workspace, it has abilities to list, read and write files.

Image Generation The agent can make calls to different image-generation services that transform a prompt into an image.

Web Search Web search is the most common application of the default agent. The agent has abilities to query a search engine and scrape website content using a headless browser.

Other Further abilities are present in the default agent. The user can be prompted to give more information about the task, the system time can be retrieved, and a completion ability is implemented.

Since the development of the default agent started, different abilities have been present and were then removed again. For example, efforts were made to introduce long-term memory to the agent. There were abilities to save agent conversations into a database for later retrieval. The idea was to improve the agent's performance by using past conversations as context. However, experiments showed that using past conversations rather hindered its capabilities. This is only an example of the constant development and changing constraints one has to deal with when working with the default agent.

Agent Structure

The AutoGPT agent is modeled after the classic agent architecture. After the start, the user is asked to enter a task the AutoGPT agent should perform. Then the agent enters a loop of prompting the LLM, executing the proposed action handling the result of the action and updating the agent state.

The LLM is prompted in a structured way. A base prompt template is defined and populated with current information before each prompting step. The information includes the task at hand, a list of possible actions, a history of previous actions and their results and some extra statements that are there to guide the language model. As the answer needs to be parsed, the system prompt defines a fixed format the LLM should answer in. The answer consists of the thoughts and the proposed next action.

AutoGPT is divided into four modules. The *emph* is the main module that controls the agent. In AutoGPT this is realized by prompting the language model in a structured way. Using the chat system prompt, the language model is prompted to answer in a structured format. Different techniques are implemented in this structure. The language model is forced not only to plan the next step but also to explain the choice for the chosen step and to add self-criticism. An extra output for the human user is also returned. The second part of the answer is the actual next action with the needed arguments. The action makes up the second module of the agent. In this module, the abilities of the agent are defined. These can be file operations, database queries or web search functionalities. The third module is the *memory*. Memories are modeled after humans which have short and long-term memory. Short-term memory can be implemented as an in-memory list of messages to the language model. Long-term memory needs persistent storage such as a

database. A popular option for language models is vector databases that work with embeddings.

Currently, the AutoGPT agent is implemented for OpenAI GPT models. The prompts are tailored in ways that benefit the characteristics of GPT-3 and GPT-4. Switching to a different model is not difficult from a software perspective, but semantically poses a challenge. If one knows the message format for a specific model the input can be adjusted. However, the same prompting techniques do not necessarily work for all language models. The challenge is to switch between different prompting styles, as every model needs to be prompted differently.

3.2 The Forge Agent

3.3 The Benchmark System

To evaluate AutoGPT and other agent systems that implement the agent protocol, the AutoGPT project has implemented a benchmarking system. The system consists of a set of tasks that the agent has to complete. The tasks are designed to test different aspects of the agent and are divided into different topics. Some tasks depend on the previous successful completion of other tasks. A task consists of an input prompt and an expected output. The output is defined by certain words that should be contained.

- built for the hackathon
- external agents can be tested
- weak documentation
- stale development

3.4 Project Overview

Originally, the default agent lived in the computer terminal and was controlled through the command line. Later, an option to start a server that serves the agent protocol was added. The user can interact with the agent through a GUI frontend running in the browser.

The web interface lists all conversations an agent had and provides a chat interface for the selected one. When the user sends an input to the agent, a task or step request is made to the server running the agent. Furthermore, the web interface can be used to start benchmarks for an agent. A single test or a complete test suite consisting of stages can be started. The GUI has no support for uploading documents to an agent. Because of this, documents for information retrieval would need to be placed in specific folder locations such that the agent can access them before it is started.

AutoGPT agents use workspaces in which they can act. Each agent has its workspace and can not modify files outside it by default. If the user wants the agent to have access to documents he needs to place them into its workspace by hand. Often, the workspace is also used to save intermediate information to text files. Some abilities produce large outputs, and the limited context window of large language models can be exceeded quickly if all intermediate information is appended to the prompt.

3.5 AutoGPT for Information Retrieval

The AutoGPT Agent has different abilities that can be utilized for information retrieval. It can search the web and operate on files, execute code and

The web search is implemented by a two-step process. First, a search API like DuckDuckGo is called to get a list of relevant pages. Then the page contents are scraped with a headless browser. It is possible to read and write text files. Other document types are processed by basic text extraction tools to get the plain text.

For longer files such as scientific journals the extracted text is too long for the language model. The AutoGPT agent cannot chunk the text into smaller chunks or store it in a database. This is a limitation that needs to be addressed for information retrieval tasks over a research database repository.

Having a vector database would enable techniques such as retrieval augmented generation. The agent would get a prompt with a question over the RDR and choose an action to start a semantic search over the vector database. The result of the search is the chunks that are semantically closest to the question. These chunks can then be included as context for the LLM prompt to generate an answer.

The default agent tends to search the web for information. We want an agent that prioritizes information that is present in the research repository. This needs to be addressed in the prompting techniques of the agent.

4

Retrieval Augmented Generation Agent

Large language models are excellent at generating text in a variety of styles. But when prompted about specific topics the answer quality suffers. Different approaches try to tackle this problem. A promising method is retrieval augmented generation, which combines in-context learning prompting techniques with a vector database. In this section, I present an agent that is designed to answer user prompts in a retrieval-augmented fashion.

To create the information retrieval agent, I used the Forge SDK [2] that is included in AutoGPT. The Forge agent SDK is a tool to create a custom agent without having to write the boilerplate code. In comparison with the AutoGPT agent, it comes with less predefined abilities and has no initial logic. On the contrary, fewer parts can break, and it is not under an ongoing re-factoring process, compared to the AutoGPT agent. Additionally, the AutoGPT agent is not optimized for information retrieval over a set of documents but operates more as a general-purpose agent.

I extended the Forge agent with abilities that are needed for retrieval augmented generation. To store the documents for retrieval, I used a *Chroma* vector database. The 'ingest' ability of the agent can be used to ingest a PDF file that is saved in the workspace. The PDF is converted into plain text and then chunked into smaller documents using the *SentenceSplitter* from *LLamaIndex* [14], which produces chunks of roughly equal length while respecting sentence boundaries.

4.1 Methods To Improve LLM Generation

Different approaches try to embed information sources into the generated text. One approach is to fine-tune the language model on a dataset that contains the information. The creation of fine-tuning data is an expensive task, as data needs to be gathered and cleaned to produce good results in training. There is some emerging work on generating synthetic datasets by using large language models as data augmentation tools. But even if the fine-tuning data quality is sufficient, it is still a challenge to make an LLM expert in a specific domain. Rather than learning new knowledge, fine-tuning is best suited to guide the model toward a certain answering style. Furthermore, fine-tuning models with billions of parameters is only possible on expensive hardware and therefore not feasible

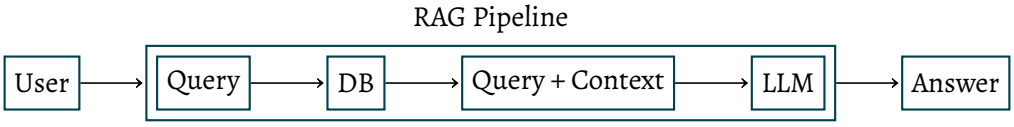

Figure 4.1: Instead of directly answering the user prompt with an LLM, a retrieval augmentation system leverages context to generate better answers. After receiving the user prompt, it is used to retrieve relevant documents from a database. These documents are appended to the language model prompt as context. The model is instructed to answer only using the provided context.

for on-demand tasks.

Following up on the challenges of fine-tuning researchers have searched for ways to optimize prompts to get the best model performance. These methods make use of a phenomenon called in-context learning. In-context learning describes the observation that giving a large language model more context surrounding the prompt can drastically improve the response quality. [29] found that adding a sentence that suggests step-by-step thinking to solve a problem makes the model better at solving logical questions. For GPT models, [31] showed an improvement after giving the language model a hint that it is an expert in a certain topic. The findings in the area of prompting techniques and in-context learning are suggesting that prompt optimizations have a high potential of getting the most out of large language models.

A more promising approach is retrieval augmented generation [12]. Before generating an answer to a prompt, a vector database is searched for relevant information. The prompt then includes the information of the returned documents as context.

4.2 Retrieval Augmented Generation

Large language models can embed a large amount of knowledge into the weights during the pre-training phase. It is possible to generate good answers for different kinds of prompts. But when it comes to specific domain knowledge, the exactness of large language models starts to degrade.

When trying to prompt models about very specific topics common problems such as hallucinations start to show. Not only does the model generate wrong information, it does so with strong confidence. A promising method to make the model answer based on specific sources is called retrieval augmented generation (RAG).

The RAG method combines an information store with in-context learning. A corpus of documents is stored in a database. When the user prompts the system the database is queried with that prompt matching documents are returned. These documents are then included in the prompt to the LLM as context.

To store the information a vector database is used. A vector database uses embedding models to embed each document into a high-dimensional vector space. After a user query, the query string gets embedded by the same model, and then a metric such as the cosine similarity is used to find semantically close documents. The top-k documents are

then returned.

4.3 Agent

I will first explain the default Forge agent. Then will describe how I built the information retrieval agent.

The Forge Agent

To make collaboration with agents easier, the open-source community created the agent protocol. The Agent Protocol defines an API schema that handles the communication with an agent. On a high level, the protocol defines endpoints to create a task and to trigger the next step for the task. The two important concepts of the agent protocol are tasks and steps:

Task A task describes a goal for the agent. A task has an input prompt and contains a list of steps.

Step A step describes a single action of the agent. A step can have custom input or copy the task input. Additionally, there is a variable that signals if this is the last step. If this variable is false, then the next step is requested automatically after completing the current. Every step has to be linked to a parent task.

The Forge SDK handles the boilerplate code that implements the agent protocol. On running, the server with the corresponding endpoints gets started, and the agent can be used over the API endpoints. AutoGPT also comes with a chatbot web app that builds the appropriate HTTP requests to the agent endpoints. The user of the Forge SDK has to create the actual agent logic, create custom prompt templates for the used model and add abilities to interact with external resources. Because the Forge SDK is still under development and by no means a polished product, some internals also need to be tweaked to achieve the desired agent behavior.

By default, the Forge agent comes with abilities to read and write text files and to search a search engine as well as scraping a webpage. Each ability is specific by a name that the LLM can use to call it. Additionally, a short description of what the ability does, input parameters and return types are described. The information about an ability is formatted into a string before adding it to the step prompt.

File System The file system abilities allow operations on files located in the workspace of the agent. The agent can only operate in the defined workspace, this prevents unwanted effects when the agent proposes unexpected actions. Abilities to read, write and list files are present. All the abilities have a path parameter that the large language model has to populate when generating a step proposal with actions from this category.

Web The web search functionality is split up into abilities to call a search engine and to read a webpage The search engine ability requires a search string that is sent to the engine. For the web page ability a URL is needed, and if specific information should be extracted the LLM also has to provide a question.

```
Reply only in JSON with the following format:

{
      \"thoughts\": {
            \"text\":  \"thoughts\",
            \"reasoning\": \"reasoning behind thoughts\",
            \"plan\": \"- short bulleted\
                        - list that conveys\
                        - long-term plan\",
            \"criticism\": \"constructive self-criticism\",
            \"speak\": \"thoughts summary to say to user\",
      },
      \"ability\": {
            \"name\": \"ability name\",
            \"args\": {
                  \"arg1\": \"value1", etc...
            }
      }
}
```

Listing 4.2: The system format of the Forge agent. The language model is asked to only answer in this format. The thoughts before creating the output for the user (speak), the LLM generates reasoning, a plan and criticism. After the model generated its thoughts, it generates an ability proposal with the corresponding arguments.

Finish The "finish" ability terminates the agent loop. The agent is asked to choose this ability if the initial user prompt can be answered and give a reason.

The Forge agent comes with a built-in template engine that is used to populate the prompts before sending them to the large language model. Some templates are included by default:

Task-Step This template is used to create the prompt that is sent to the language model for each step. It includes the current task description and placeholders for extra information. The template always needs to be populated with the list of available abilities, allowing the language model to choose one of them.

System-Format The system format defines how the model should respond to prompts. The Forge system format is depicted in Listing 4.2. The responses of the model need to be parsed according to this definition. Language models have different capabilities in answering in a structured manner, so the format has to be tuned for every model.

Techniques Techniques is a collection of prompting techniques that have been shown to improve generation quality. By default, the Forge agent has templates for few-shot, expert and chain-of-thought prompting.

The core of an agent is its logic. The Forge agent comes without any logic, its default behavior is to write a boilerplate text into a file in the workspace.

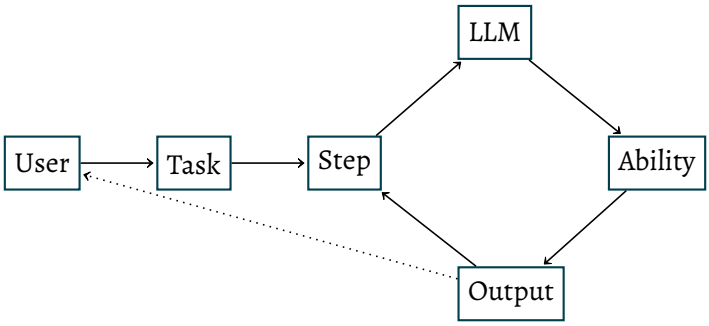

Figure 4.3: The information retrieval agent receives the user prompt at the start of each cycle. Next, the step prompt is constructed with the action history, list of abilities and extra information. After the LLM responds to the step prompt, the proposed action is executed and the step output is presented to the user. The loop ends if the LLM proposes the 'finish' ability.

Agent Memory

Similar to the view of human memory, different implementations of memories for agents have been proposed. For short-term memory, the message history format used by instruct language models can be used. The last n messages of a conversation between the user and the agent are recorded and change the generation behavior of the LLM that controls the agent. For example, a researcher could

Long-term memory is generally represented as some external store where information can be stored and retrieved. This memory can include past conversations between the agent and the user, as well as additional data that the agent should have access to. The information can be present from the beginning or added at runtime by a user prompt.

For large language model applications vector databases are a popular choice, because they integrate well into the embedding-based language modeling concept. Vector databases store data together with their high-dimensional embedding. Embedding models that are specifically trained to produce rich embeddings are used. The agent uses a *chromadb* vector database [6].

Agent Main Loop

The agent program is defined in the agent step execution function. Every time a new step request is given this function is called. For the first user input, a task is created, then the agent proceeds to complete steps until the task goal is reached. A single cycle of the agent follows a fixed set of actions. First, the step input is updated for the current step. The user can give updated instructions in each step to further guide the agent. If no new input is given, the agent uses the input of the parent task as the step input.

After the input is updated the agent constructs the message history for the language model. The first message is always the same system format specification. The agent uses the default Forge agent format shown in Listing 4.2. The second message is the populated step prompt, which describes the current perceived environment of the agent. When the

step prompt is constructed, it is sent to the language model. The generated answer is parsed into the thoughts and ability parts.

The response is parsed into the thoughts of the LLM and the proposed ability. If a valid ability is proposed, it is executed.

Step Prompt

The main step of the agent is to populate its step prompt template As denoted in section 2.2, an agent perceives its environment through sensors. Our agent describes the current environment in the step prompt. An example step prompt is shown in Listing 4.4. It includes the current step input and instructions on how to answer. The available resources are denoted, and the action history is appended.

If the agent is in its first step, the action history is empty. Otherwise, a list of the previous actions is compiled to give the agent a sense of its current state inside the task context. Each entry has the proposed ability with its parameters and the ability output. If the ability produced an error, this is also denoted. In the best practices section, the agent is prompted to react to errors and to not use the same action with the same arguments again.

In addition to the action history, the available abilities are added to the prompt. The abilities enable the agent to act out its decisions in the environment. For each ability, the name, parameters, return type and a short description is given.

Abilities

In section 2.2, I described how agents use actuators to perform actions in the environment. The IR agent is a software agent that uses functions to modify its workspace. In particular, it has abilities that enable retrieval augmentation capabilities. These abilities will be described in the following.

Ingest The *Ingest* ability should be used when a document should be a source of information for the agent. A file with the specified filename has to exist in the agent workspace before calling this function. This ability converts the PDF into plain text, creates text chunks of the same length and calls the functions to embed them into the vector database.

Retrieve The *Retrieve* ability accepts to query string and an output file path. With the query string, the agent queries its memory and gets the most relevant documents for it. The documents are formatted and saved to the specified file path.

Answer This ability uses the contents of a text file to populate the augmented generation template. The populated template is then sent to the LLM to generate an answer based on that context.

As the LLM tends to choose web search abilities to collect information, I removed the corresponding abilities. The agent is now forced to retrieve information from local sources.

```
Answer as an Expert in Planner.
Your task is:

What characterized prompt book usage in the late eighteenth
century?

Answer in the provided format.

Your decisions must always be made independently without seeking user
    assistance. Play to your strengths as an LLM and pursue simple
    strategies with no legal complications.

## Resources

You can leverage access to the following resources:

- A vector database representing your memory. You can ingest documents into
    it and query it for relevant information.

## Abilities

You have access to the following abilities you can call:

- answer_with_context(prompt: str, context_file: str, output_file: str) ->
    None. Usage: Answer a prompt using content
from a textfile as context,

- retrieve_context_from_memory(query: string, output_file: string) ->
    string. Usage: Retrieve the most relevant information for a query and
    save it to a .txt file,

- finish(reason: string) -> None. Usage: Use this to shut down once you
    have accomplished all of your goals, or when there are insurmountable
    problems that make it impossible for you to finish your task.

...
## Best practices

- Prefer querying your memory to retrieve source before answering.

...
```

Listing 4.4: The step prompt is the core of each step the agent takes. It describes the current environment representation of the agent. First, the current task is presented to the agent. Next, available resources and abilities are described. Finally, some hints for best practices are denoted. In this example, there is no step history as this is the first step. A full step prompt example is show in

```
These manuscripts are significant evidence of the different aspects of a
||
5 Chardonnens 2007.


Introduction | 7

performance and its adaptability to different periods and contexts. A manu-
script of this kind which is produced communally prepares and shapes the
    col-
lective performance.
Two different aspects of the temporality of a performance are highlighted by
Mathieu Ossendrijver in his article on Babylonian and Assyrian sources pro-
duced during the first millennium BCE.
```

Listing 4.5: A chunk produced by the SentenceSplitter that spans two pages. The converted PDF plain text contains page titles, numbers, footnotes and other additional information besides the main content. The information splits the article content into multiple parts. This can hinder the quality of the produced embeddings.

Document Ingestion

To enable retrieval augmented generation, the agent can ingest documents into its vector database. The *ingest* ability can be called with a PDF filename as the argument. For the ability to succeed, a PDF file with the exact filename has to be present in the agent workspace. The user has to manually place the file in the workspace before prompting the agent. If a PDF is found, it is converted into plain text. Then, the text is split up into smaller chunks that can be mapped to embeddings with an embedding model. The *SentenceSplitter* utility from the *LLamaIndex* package [14] is used, to create the chunks. The splitter splits up a text recursively and then joins words back together to generate chunks of similar length while respecting sentence borders.

As PDF is an unstructured format, the conversation to plain text is not trivial. In addition to the main content, the article contains extra information like page numbers, references or footnotes. These extra elements can split paragraphs that span two pages. After splitting up the text with the SentenceSplitter, these elements are still present in the chunks. An exemplary chunk is shown in listing 4.5. For the following embedding step, this can hinder the semantic accuracy of the embedding. After splitting the text into chunks, they are added to the *ChromaDB* vector database [6].

Language Model Response

The populated step prompt is sent to the language model. To react to the model answer, the agent parses the generated response. From the thoughts part of the answer, only the 'speak' string is used as an output to the user. The second part contains the action proposal for the step.

5

Benchmarking for an IR Agent

The evaluation of large language model agents is a difficult task, as evaluating LLMs themselves presents a challenge. There are different approaches to evaluating systems built around language models. Subjective evaluation is based on human feedback. As LLM systems are generally made to serve humans this is an important part of evaluation. On the other hand, quantitative metrics that can be computed are used for objective evaluation. Different metrics are used for different tasks. Another important method is benchmarks. Benchmarks are a set of tasks or an environment that the agent is to move in.

When talking about agents, different properties can be evaluated. LLM Agents can be evaluated in an end-to-end fashion, where only the output is relevant for measuring performance. Other methods also use intermediate outputs to evaluate agent decisions on a step level.

5.1 Existing IR Agent Benchmarks

As the space of possible agent domains is large, lots of different benchmarks were proposed. Simulation environments like

Although lots of benchmarks for LLM applications have been proposed, few benchmarks are designed to test the information retrieval capabilities of an agent. Some benchmarks are used to evaluate information retrieval in general and in diverse domains. In [16] introduced a general benchmark for AI assistants.

5.2 Custom Benchmarks for Local IR over Journals

Large language models have delivered good results for general knowledge questions, but can struggle with domain-specific tasks. They can confidently generate answers that are completely made up but seem correct to a non-expert user. For this reason, the IR agent uses retrieval augmented generation to gather context from a local vector database, before generating an answer. The retrieval does not have to be evaluated, as the agent uses an external library that is tested separately. What needs to be tested is the answer quality of the chatbot. As the built-in benchmarking system of AutoGPT was unfinished and not

ready to be used for custom agents, I had to test question-and-answer pairs by hand in order to evaluate the IR agent. I adapt the end-to-end evaluation based on cosine similarity proposed in [3], to compare the IR agent to other chatbot services.

Humanities Journal for Expert Knowledge

We need documents that contain domain-specific expert knowledge to evaluate the answer quality of the information retrieval agent. To generate question-and-answer pairs, I used the humanities journal *Manuscripts and Performances in Religions, Arts and Sciences* [4]. The journal presents a collection of papers that offer perspectives on how manuscripts can be studied as objects and actors in various kinds of performances. From the journal, I handpicked 10 question and answer pairs for evaluation. While this set of pairs can not be used to evaluate the general question and answer pairs, it can be used to compare the IR agent to other services.

The questions in the dataset follow a certain pattern. In particular, they ask for specific facts from the journal articles. In order to give the correct answer, one has to know the journal.

Comparing with other LLM IR Options

To put the evaluation results into perspective, other services for information retrieval were tested on the Q&A dataset. I tested GPT-3.5-Turbo, GPT-4-Turbo and Perplexity for comparison. For all tools, the answer quality was measured by calculating the cosine similarity between their embeddings. I used the OpenAI Embeddings API [9] to generate the embeddings of all questions and model answers. All services are only receiving the question with any additional information.

6

Results

While working with AutoGPT and developing the information retrieval agent, a couple of weaknesses of current LLM agent systems became visible. The most important challenge for LLM agents is the non-deterministic generation of large language models. It makes it significantly harder to build a software system around it because every time the language model is called, there has to be a fallback mechanism in case something fails. For fixed pipeline systems this is true as well, but error handling is easier as the task steps are known beforehand.

6.1 Points of Failure

Using large language models as agent controllers is a recent idea, and therefore there is a lot of experimentation left to do. A challenge when developing such agents is the natural language interface that language models communicate with. While it makes describing the task easier for humans, it complicates the internal communication in the software. For example, AutoGPT wants a certain system format that the LLM should respond in. While this works most of the time, it is not guaranteed that the answer complies with the format. A small deviation such as a missing bracket can break the parsing process. Therefore, a lot of effort and time is put into creating better tools and frameworks that handle such mistakes, instead of doing actual research on the agent performance.

Another aspect is the non-deterministic generation of large language models. For the same prompt, large language models can generate a different answer. This answer will probably be semantically similar to the previous one, but when a model has to choose between two abilities, this small difference can decide if the agent succeeds or fails at completing the task. Like earlier, developers have to spend time handling these cases by re-prompting several times or putting fallback actions in place.

For a lot of tasks, a static pipeline is enough to meet the user demands. Almost all retrieval tools that work in production use a fixed pipeline.

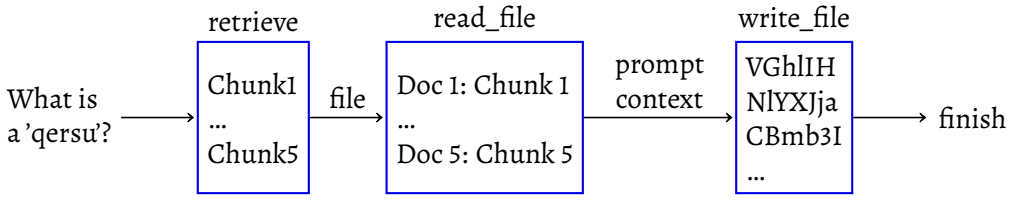

Figure 6.1: An agent run that failed because of an incorrect action choice. The resulting answer is just random characters.

Wrong Ability Choice

The agent sometimes chose abilities that were not suited to progress towards the goal. When the agent had read and write abilities present, sometimes the agent chose to read the file containing the retrieved context for a query. The correct choice would have been to process with the Answer ability. In some instances, choosing the wrong ability leads to useless answers, as shown in figure 6.1.

A solution to this problem is to either remove the read and write abilities or to guide the agent with explicit instruction in the prompt on which ability to use. For a true autonomous agent, these solutions are not ideal, because we want the agent to make the decisions with minimal user intervention. Fixes like removing abilities bring the agent closer to fixed pipeline solutions. Another solution could be to retry the same or modified prompts. This is feasible if the user does not demand fast responses. In the case of information retrieval, the agent should answer reasonably fast and give correct answers. Therefore, I opted to remove all abilities that were not needed for retrieval augmented generation.

Furthermore, the model choice is important for decision-making. GPT-4 is a lot better than GPT-3 at making good decisions, following tasks and providing criticism and reasoning. For this reason, GPT-4 was used as the agent controller LLM.

Wrong Answer Format

After receiving the answer to the step prompt, the agent has to parse it. The parsing step can only work if the answer is generated in the exact format specification. It is not certain that an LLM adheres to the system format. The capabilities in answering in a structured format given by the user differ between models. Larger models like GPT-4 are better than smaller ones like GPT-3, but also cost more. Other models improve at structured answers with fine-tuning but might lose capabilities in other areas as a result.

There are solutions, that try to catch common mistakes from LLMs such as missing brackets or missing commas in JSON. While some mistakes can be mitigated with this approach, it is not feasible for large-scale applications as there are too many where the format can break. Similar to the decision-making mistakes, another solution is to retry and hope for a parsable answer.

6.2 Benchmarking Results

6.3 Subjective Evaluation

7

Conclusion

Using language model agents for information retrieval tasks is an interesting approach, but needs to overcome some key challenges for usage in real-world applications.

7.1 Future Work

The research area about large language models currently moves at a rapid pace. While writing this thesis, communities have focused more on agent applications of LLMs. OpenAI has released OpenAI Assistants, which.... The AutoGPT team is still working on making the use of local models feasible. In this work, only information retrieval over local documents was done. This could be extended with web searching capabilities to include external information on demand. For retrieval augmented generation, the ingestion of documents into the vector database is a crucial step. The popular chunking methods for unstructured data like PDFs simply split the text to get a certain chunk size. Methods for semantically splitting up unstructured data might improve the performance of retrieval augmentation pipelines.

Bibliography

[1] *Assistants Overview - OpenAI API*. (Visited on 03/20/2024).

[2] *AutoGPT Forge Github*. (Visited on 03/01/2024).

[3] Banerjee, D., Singh, P., Avadhanam, A., and Srivastava, S. *Benchmarking LLM Powered Chatbots: Methods and Metrics*. Aug. 8, 2023. DOI: 10.48550/arXiv.2308.04624. arXiv: 2308.04624 [cs]. (Visited on 03/21/2024). preprint.

[4] Brita, A., Karolewski, J., Husson, M., Miolo, L., and Wimmer, H., eds. *Manuscripts and Performances in Religions, Arts, and Sciences*. De Gruyter, Nov. 20, 2023. ISBN: 978-3-11-134355-6. DOI: 10.1515/9783111343556. (Visited on 02/29/2024).

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language Models Are Few-Shot Learners. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. (Visited on 03/08/2024).

[6] *ChromaDB*. (Visited on 03/18/2024).

[7] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. Dec. 11, 2014. DOI: 10.48550/arXiv.1412.3555. arXiv: 1412.3555 [cs]. (Visited on 03/06/2024). preprint.

[8] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *Proceedings of NAACL-HLT*. 2019.

[9] *Embeddings - OpenAI API*. (Visited on 03/22/2024).

[10] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. *Convolutional Sequence to Sequence Learning*. July 24, 2017. DOI: 10.48550/arXiv.1705.03122. arXiv: 1705.03122 [cs]. (Visited on 03/06/2024). preprint.

[11] *Introduction - AutoGPT Documentation*. (Visited on 03/19/2024).

[12] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[13] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. *Lost in the Middle: How Language Models Use Long Contexts*. Nov. 20, 2023. DOI: 10.48550/arXiv.2307.03172. arXiv: 2307.03172 [cs]. (Visited on 02/29/2024). preprint.

[14] *LlamaIndex SentenceSplitter*. (Visited on 03/24/2024).

[15] Manning, C., Raghavan, P., and Schuetze, H. *Introduction to Information Retrieval*. Online edition. Cambridge University Press, Apr. 2009. ISBN: 0-521-86571-9.

[16] Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. *GAIA: A Benchmark for General AI Assistants*. 2023-11-21, 2023. DOI: 10.48550/ARXIV.2311.12983. arXiv: 2311.12983 [cs.CL]. preprint.

[17] Nakajima, Y. *Yoheinakajima/Babyagi*. (Visited on 03/08/2024).

[18] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training Language Models to Follow Instructions with Human Feedback. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 27730–27744.

[19] Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative Agents: Interactive Simulacra of Human Behavior. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST'23. ACM, Oct. 2023. DOI: 10.1145/3586183.3606763.

[20] *Perplexity AI*. Perplexity AI. (Visited on 03/20/2024).

[21] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. *Improving Language Understanding by Generative Pre-Training*. OpenAI, 2018.

[22] Robertson, S. and Zaragoza, H. The Probabilistic Relevance Framework: BM25 and Beyond. In: *Foundations and Trends in Information Retrieval* 3(4):333–389, Apr. 1, 2009. ISSN: 1554-0669. DOI: 10.1561/1500000019. (Visited on 03/06/2024).

[23] Russel, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Fourth Edition, global edition. Pearson, 2022. ISBN: 978-1-292-40113-3.

[24] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. *Toolformer: Language Models Can Teach Themselves to Use Tools*. Feb. 9, 2023. DOI: 10.48550/arXiv.2302.04761. arXiv: 2302.04761 [cs]. (Visited on 03/18/2024). preprint.

[25] Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. *HuggingGPT: Solving AI Tasks with ChatGPT and Its Friends in Hugging Face*. Dec. 3, 2023. DOI: 10.48550/arXiv.2303.17580. arXiv: 2303.17580 [cs]. (Visited on 03/18/2024). preprint.

[26] Significant Gravitas *AutoGPT*. (Visited on 03/18/2024).

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017.

[28] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. *A Survey on Large Language Model Based Autonomous Agents*. Aug. 2023. DOI: 10.48550/ARXIV.2308.11432. arXiv: 2308.11432 [cs.AI]. preprint.

[29] Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 24824–24837.

[30] Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al. *The Rise and Potential of Large Language Model Based Agents: A Survey*.

Sept. 19, 2023. DOI: 10.48550/arXiv.2309.07864. arXiv: 2309.07864 [cs]. (Visited on 02/29/2024). preprint.

[31] Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., and Mao, Z. *ExpertPrompting: Instructing Large Language Models to Be Distinguished Experts*. May 23, 2023. DOI: 10.48550/arXiv.2305.14688. arXiv: 2305.14688 [cs]. (Visited on 03/14/2024). preprint.

[32] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In: *Advances in Neural Information Processing Systems* 36:11809–11822, Dec. 15, 2023. (Visited on 03/20/2024).

[33] Zhu, Y., Yuan, H., Wang, S., Liu, J., Liu, W., Deng, C., Chen, H., Dou, Z., and Wen, J.-R. *Large Language Models for Information Retrieval: A Survey*. Jan. 19, 2024. DOI: 10.48550/arXiv.2308.07107. arXiv: 2308.07107 [cs]. (Visited on 03/06/2024). preprint.