Aalto-yliopisto
Aalto-universitetet
Aalto University

ELEC-E7852
Computational Interaction and Design

Assignment A4a

Generative Agents with LLMs

Aitor Urruticoechea Puig
aitor.urruticoecheapuig@aalto.fi
Student N°101444219
November 2024

Assignment A4a
Aitor Urruticoechea Puig
November 2024

Acknowledgement

Parts of the code implemented in this work were generated with the assistance of GitHub Copilot. None of its proposals were, however, employed without refinement and necessary tweaks to adapt it to the nuances of the tasks at hand; making it impossible to identify and subsequently mark which lines of code were human or machine-made, for many were the fruit of the combination of both.

Contents

List of Figures

1 Introduction and Set Up

The goal for this assignment is to develop and implement a generative reading agent that can interact and serve as a counterpart to the provided reading assistant, which was also tested in class (see Figure 1).

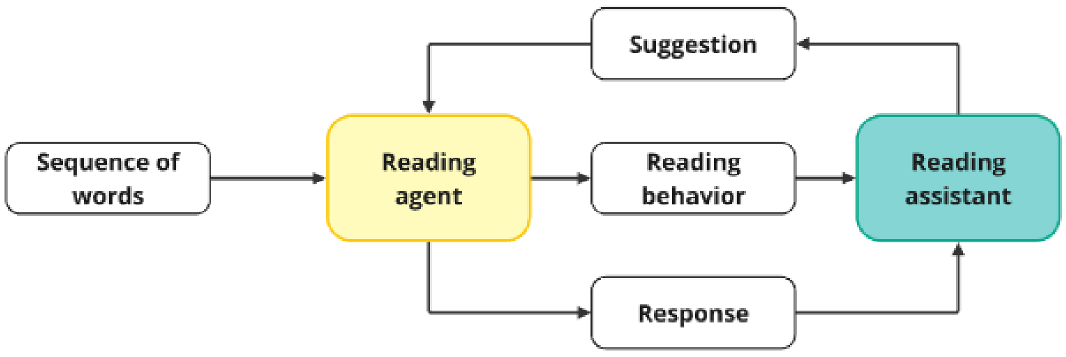

Figure 1: System flowchart. The task is to develop and implement the box in yellow.

To that end, a very basic system has been set up as a starting ground, where a PDF-type file is read (using the library PyPDF2) and processed into a list of strings, each representing a sentence that the agent will later be able to read through. Similarly, before getting into the agent itself, it is also worth discussing two other initializations.

First, the "agent memory", which in .json format will allow for the recording and automatic retrieving of the actions the agent takes; and can serve for later analysis. This reading stream will record which actions the agent takes (reading behaviour), what sentences have been read (read sentences), what reflections have been done on the read information (reflections), what suggestions from the assistant have been followed (suggestions followed), and what have these yielded (achieved from suggestion). More on how all these are generated later, but for now it is worth knowing the method in which they will be managed.

```
reading_stream = {
    "reading_behavior": [],
    "read_sentences": [],
    "reflections": [],
    "suggestions_followed": [],
    "achieved_from_suggestion": []
}
```

Second, the learning goals that the agent needs to achieve will need to be generated. This will later be used as a guideline for generating the agent's actions, as well as a point to compare what has been achieved so far, eventually serving as a stopping device once the agent has learnt everything they needed from the provided document. This will be generated with relatively simple prompt engineering, taking as main inputs the name of the file and the first sentence of it, which is assumed to be the title of the document.

It is worth noting that during testing, and due to the limited contextual information the model gets from this point, the learning goals tended to be too complex for a single reading of one document; so the prompt has been toned down to "easy goals", which outputs more attainable ones.

```python
def set_knowledge_goal(path, title):
    # From the PDF title, the agent generates a goal to learn about the topic. This will be
    ↪  compared against to decide whether the agent has enough knowledge.
    prompt = f'A generative agent is set to start reading the file "{path}". Knowing this
    ↪  and the title "{title}", set a comprehensive list of goals (between 3 and 5) for
    ↪  their learning; which are attainable only by reading the document. Make them easy
    ↪  goals. Return only these goals.'
    prompts = [{
            "role": "system",
            "content": "You are an expert educator; with in-depth knowledge of how people
            ↪  and generative agents learn. Thus, you know perfectly well how to set goals
            ↪  for the different stages of learning at any level."
        },
        {
            "role": "user",
            "content": prompt
    }]
    goal = generate_response(prompts)
    return goal
```

2 Reading Agent design and implementation

The proposed reading agent is, obviously, LLM-powered, but its core behaviour is still choice-ruled. At every interaction step, it has four potential actions that it can choose to do according to context:

- **Read**: the agent can choose to simply move to the next sentence in the document, gaining access to that information.
- **Reflect**: alternatively, the agent can choose to pause the lecture and instead spend effort reflecting, interiorizing, and analysing the obtained knowledge so far. It is in this reflection action that the learning goals will be evaluated for completion.
- **Follow**: if the reading assistant has suggested something; the agent can choose to follow that suggestion in hopes of advancing its understanding towards the learning goals.
- **Other**: finally, there exists a free, open-ended option in case the agent decides to do something else. However, in no test cases this has been used extensively enough to warrant analysis.

This decision is given to the agent via prompting, and is stored, predictably, in the reading stream under reading behaviour. It is worth noting that the flow of prompts and responses is what will be used continuously to give the relevant context each time to the agent. Thus, once initialized with the system prompt and the first user prompt, the rest of the times the prompt will begin with what has been achieved by the previous action. In short, the overall prompt stream will end up looking something along the lines of this toy example (note that output specifications have been skipped):

1. *System*: {system prompt}
2. *User*: You just started your new assignment. Your goals for the current document are {goals}. What would you like to do?
3. *Assistant*: READ
4. *User*: You read the 1st sentence: {sentence}. The reading assistant has suggested {suggestion}. (...) What would you like to do?
5. *Assistant*: REFLECT

6. *User*: You reflected: {reflection}. The reading assistant has suggested {suggestion}. (...) What would you like to do?
7. (conversation continues...)

This is implemented in a general governing function, that is to be called at every step of the main loop. This function returns each time a pair of this interactions. So, when called once, will return the updated prompt history, as well as the behaviour that the action chosen has generated (more on how these behaviours are generated in the following subsections). Critically, this outputted user behaviour is what will be the reading assistant will be able to interpret in order to suggest actions; which will be used as an input in the subsequent iteration.

```python
def user_behavior(previous_prompts, goals, sentence_count, sentences, suggestion):
    goal_check = False
    if len(previous_prompts) == 0:
        prompts = [{
            "role": "system",
            "content": "[SYSTEM PROMPT - see actual code file]"
        },
        {
            "role": "user",
            "content": "[STARTING USER PROMPT - see actual code file]"
        }]
    else:
        previous_prompts[-1]["content"] += f"""\n The reading assistent has suggested:
        ↪ {suggestion} \n.
        Remeber your learning goals: {goals} \n
        What would you like to do now? (READ, REFLECT, FOLLOW, or your own behavior)"""
        prompts = previous_prompts

    decision = generate_response(prompts)
    reading_stream["reading_behavior"].append(decision)
    prompts.append({ "role": "assistant", "content": decision})

    if "READ" == decision:
        current_sentence = sentences[sentence_count]
        sentence_count += 1
        behavior = read_next_sentence(sentence_count, current_sentence)
    elif "REFLECT" == decision:
        behavior, goal_check = read_reflect(goals)
    elif "FOLLOW" == decision:
        behavior = follow_suggestion(suggestion)
    else:
        behavior = decision

    prompts.append({"role": "user", "content": behavior})
    return prompts, behavior, sentence_count, goal_check
```

Now, one can move forward to how the actions that the reader agent can take are generated according to context.

2.1 Reading action

The reading action is arguably the easiest to implement, as one only needs to properly process the sentence list that has been obtained from pre-processing the document, and correctly phrasing for both the reading stream and the prompt stream. One has to be careful, though with the edge case of having read all the document already, in which case this action should just return a default message informing of just that. The read sentences are, naturally, sstored in the reading stream for later calls.

```python
def read_next_sentence(sentence_count, current_sentence):
    if current_sentence == "!$& END OF DOCUMENT !$&" or sentence_count >= len(sentences):
        action = "The reading agent has reached the end of the document."
    else:
        reading_stream["read_sentences"].append(current_sentence)
        action =  f"The reading agent reads the '{sentence_count}' sentence:
        ↪  {current_sentence}."
    return action
```

2.2 Reflect action and goal evaluation

The reflection action actually needs to perform to separate tasks: first, it needs to, of course, reflect upon the read sentences, actions performed, and previous reflections. These reflections are recalled from the reading stream, and later updated in the same to keep the knowledge database up to date. After that, the agent needs to evaluate if the knowledge obtained by this new reflection, added to all the previously obtained knowledge, satisfies the learning goals set at the very beginning. This is done using two different calls to the model, each with carefully crafted system and user prompts. This goal evaluation yields a boolean variable that subsequently updates that status in the main user_behavior function, ultimately serving as a governing switch for the general reading agent - assistant loop.

```python
def check_enough_knowledge(goals):
    prompts = [{
        "role": "system",
        "content": "You are an expert evaluator; with the ability to assess the knowledge
        ↪  of a generative agent based on the goals set for their learning."
    },
    {
        "role": "user",
        "content": f'So far, the agent you are evaluating has read the following sentences:
        ↪  {reading_stream["read_sentences"]} \n and reflected on the knowledge like this:
        ↪  {reading_stream["reflections"]}\n'
            f'Also, the agent has achieved the following from past suggestions:
            ↪  {reading_stream["achieved_from_suggestion"]}\n'
            f'The agent has the following learning goals:\n {goals} \n.'
            f'Given all of this, has the agent, through reading, reflections, and suggested
            ↪  actions, achieved their learning goals? Be sufficiently strict: ALL the
            ↪  goals must be achieved to return YES.'
            f'Return only a YES/NO response.'
    }]
    answer = generate_response(prompts)
    answer = answer.lower().strip()
    if "yes" in answer:
        enough_knowledge = True
    else:
        enough_knowledge = False
    return enough_knowledge
```

```python
def read_reflect(goals):
    prompts = [{
        "role": "system",
        "content": "You are an autonomous generative agent designed to learn from a document you are reading."
    },
    {
        "role": "user",
        "content": f"So far, you've read the following sentences: {reading_stream['read_sentences']}\n"
          f"You've previously reflected on the knowledge: {reading_stream['reflections']}\n"
          f"Also, you've previously achieved the following from past suggestions by a reading assistant: {reading_stream['achieved_from_suggestion']}\n"
          f"The agent's goals are: {goals}\n"
          "Briefly reflect on all this information. Interiorize it, debate it, think about your goals, etc. Return only this reflection."
    }]
    reflections = generate_response(prompts)
    action = f"The reading agent reflects on the information: {reflections}"
    reading_stream["reflections"].append(reflections)

    # Check goals
    goal_check = check_enough_knowledge(goals)

    return action, goal_check
```

2.3 Follow suggestion action

Finally, for the part of following suggestions, a bit of prompt engineering is used to make the model create a plausible narration of the outcomes that can be achieved by following the action suggested by the reading assistant. This is the best found way to mimic what a real human researcher might do when following a suggestion like this. For knowledge management, the suggestions that make it this far are also stored in the reading stream, alongside the results from following these suggestions.

```python
def follow_suggestion(suggestion):
    prompts = [{
        "role": "system",
        "content": "You are an autonomous generative agent designed to learn from a document you are reading. To aid in your task, there is also a reading assistant that will, from time to time, suggest actions; though it is up to you to decide whether to follow them or not."
    }, {
        "role": "user",
        "content": f"So far, you've read the following sentences: {reading_stream['read_sentences']}\n"
            f"You've previously reflected on the knowledge: {reading_stream['reflections']}\n"
            f"Also, you've previously achieved the following from past suggestions: {reading_stream['achieved_from_suggestion']}\n"
            f"The reading assistant has suggested: {suggestion}\n"
            f"And you have decided to follow it. What actions do you do to follow the suggestion? What do you achieve and/or learn from it? Answer with the following format: The reading agent follows the suggestion given, ..."
    }]
    action = generate_response(prompts)
    reading_stream["suggestions_followed"].append(suggestion)
    reading_stream["achieved_from_suggestion"].append(action)
    return action
```

3 Results and Conclusions

The proposed approach to the reading assistant has been tested using two input PDF files. First, a more complex and lengthy one: *"A Cyborg Manifesto"* by Donna J. Haraway (University of Minnesota Press, 2016); a philosophical - political essay on technology, society, and socialist feminism. Second, a more simple, semi-technical document, the PDF submission to Assignment A1a of this same course, by the same author as this very homework. While this is a small test population, it can be argued it is good enough for the scope of this assignment, as it showcases the range, successes and limitations of the developed system.

Starting with the second test file, the agent managed to achieve the generated learning objectives quite successfully. The goal generator function dictated the following learning objectives:

- Understand the basic concepts of combinatorial optimization: Identify and define key terms and ideas related to combinatorial optimization as presented in the document.
- Analyze the structure and components of the DespesApp project: Identify how the project serves as a baseline and highlight its main components as mentioned in the assignment.
- Identify the objectives of Assignment A1a: Break down the specific aims and tasks outlined in the assignment to understand the expected outcomes.
- Recognize the connection between computational interaction, design, and combinatorial optimization: Explore how these concepts are interrelated within the context of the assignment.
- Extract key methodologies or techniques used in the assignment: Identify any specific methods or approaches utilized in addressing combinatorial optimization in the DespesApp project.

All of them were achieved approximately when the reading agent finished reading the introduction section, as can be seen in the adapted reading stream .json file, where the reflections support the decision that the learning goals were sufficiently achieved. It is interesting how, in this case, none of the suggested actions by the reading assistant were followed - they were deemed unnecessary or unhelpful. This is not always the case, as can be seen in the provided demo video, where a successful outcome is achieved but does follow one recommendation in the process.

```
{"reading_behavior": ["READ", "READ", "REFLECT", "READ", "READ", "REFLECT", "READ", "READ",
↪  "REFLECT", "READ", "READ", "REFLECT", "READ", "READ", "REFLECT", "READ", "REFLECT",
↪  "READ", "REFLECT", "READ", "REFLECT", "READ", "REFLECT"],
"read_sentences": [...],
"suggestions_followed": [],
"achieved_from_suggestion": [],
"reflections": ["The document presents an assignment centered on combinatorial optimization
↪  as part of a course in Computational Interaction and Design. One key aspect is its
↪  connection with the DespesApp project, which serves as the baseline. (...)",
↪  "Reflecting on the document, several aspects emerge that align with the agent's goals.
↪  First, the assignment is situated within the context of combinatorial optimization and
↪  computational interaction, tied closely to the DespesApp project. (...) Ultimately, the
↪  aim is to extract techniques and methodologies from this assignment to gain insights
↪  into advanced design and optimization practices, reinforcing the idea that technical
↪  proficiency must coexist with user-centered design principles for meaningful
↪  innovation.", "(...) it's clear that the assignment is deeply rooted in the principles
↪  of combinatorial optimization and computational interaction, (...) As I delve deeper
↪  into the assignment's methodologies, the focus will be on extracting practical
↪  techniques that demonstrate advanced design and optimization practices", "Combinatorial
↪  optimization, typically involving algorithmic techniques to address complex problems,
↪  is juxtaposed with the user-centered approach of computational interaction and design
↪  within the DespesApp framework. The DespesApp project, serving as the practical
↪  foundation, provides a tangible example of how these theories are applied. It's a
↪  spending tracking app, (...). This environment encourages exploration of innovative
↪  problem-solving approaches, potentially enhancing both the assignment's execution and
↪  broader collaborative opportunities.", "(...) the assignment appears to involve
↪  optimizing the UI by addressing layout complexities and improving interaction, needing
↪  a blend of technical proficiency and design thinking. (...) Examining these
↪  interconnections not only highlights practical methodologies employed in the assignment
↪  but underscores the importance of integrating technical and creative processes. (...)",
↪  "(...)", "(...)", "(...) By identifying specific methodologies employed in the
↪  assignment, deeper insights can be gained into practical applications of these abstract
↪  concepts.\n\nApproaching the task through a lens that appreciates the synergy between
↪  technology and creativity is instrumental. It reinforces the importance of advancing
↪  both technical and creative design proficiencies, ensuring that computational design
↪  and optimization are not only theoretically sound but also practically effective. This
↪  reflection highlights the importance of a comprehensive approach to tackling complex
↪  design challenges, fostering innovation that is both user-centered and technically
↪  robust."]
}
```

On the flip side, the first test file was not as successful. Be it due to the complexity of the text, the complexity arising from the objectives that get defined by such a well-known title, or due to the length of the document, which in turn generates very length prompts and creating a domino effect of actions down the line; the reading assistant was not successful in achieving the learning goals. The sentence and word complexity used by Haraway may be of interest for further analysis, as the agent considers they need reflection very frequently. With the exception of sentences not belonging to the text itself (subtitles, footnotes, etc), the agent repeatedly decides to stop after every sentence to ponder and reflect. This is quite unhelpful, as not advancing quickly enough through the text makes the reflections unusable in advancing towards the learning objectives, and rather become an eco-chamber of the agent's own thoughts.

```
{"reading_behavior": ["READ", "READ", "READ", "READ", "READ", "REFLECT", "READ", "READ",
↪  "READ", "REFLECT", "READ", "READ", "REFLECT", "READ", "READ", "REFLECT", "READ",
↪  "REFLECT"],
"read_sentences": [...],
"reflections": [...],
"suggestions_followed": [],
"achieved_from_suggestion": []}
```

Objective definition and evaluation is also worth analysing more thoughtfully. The provided video example, where this last output was extracted from, showcases an uncommon yet occurring case of false positive when evaluating learning. Rather than noticing that the reading agent is just reflecting based on very little actual information from the document, the evaluator decides that the reflections are good enough to have a passing grade when checking against learning objectives.

In conclusion, the results showcase both the strengths and limitations of the proposed reading agent. On the one hand, the system demonstrates significant success with the simpler, semi-technical document, achieving all learning objectives effectively either by following the reading assistant or by itself. On the other hand, the agent's performance with the more complex and lengthy text of A Cyborg Manifesto was markedly less successful. The challenges posed by the document's complex language, extended length, and the demanding nature of its associated learning objectives showcase critical limitations in the system's current implementation. Frequent and repetitive reflections slowed progress significantly. Additionally, false positives in evaluating learning outcomes shed light into areas that definitely would benefit from refinement in the system. Despite all this, the agent's success with the simpler document highlights its potential for structured, goal-driven reading scenarios, offering a foundation for potential future developments.