Causal Graph Identification by LLMs

The primary goal of the project is to conduct an empirical study to assess the possibility of performing causal analysis using Large Language Models (LLMs).

The project focuses on the operation of causal discovery, which is the task of learning the structure of causal relationships between variables and entities; its output is a directed graph that represents the underlying data-generation process (DGPO) and provides insight into the true causal relationships between variables. The generated graph is used as a base for many other (if not all) fundamental tasks in causal analysis (e.g. effect inference, prediction, attribution) [1].

The causal discovery task is performed starting from natural language, that is textual data, such as scientific papers and research publications.

The data is then processed to extract the main textual entities; a naïve discovery procedure is then used to find the causal relationship between these entities.

The final operation creates the causal graph using the causal relationships found in the previous step and plots the directed acyclic graph (DAG).

The project can be divided in two main steps: data collection and data analysis. The former one consisted in collecting the necessary data for the latter, which can itself be divided into multiple other sub-operations.

First, the data collection process will be presented, highlighting the utilization of the National Center for Biotechnology Information (NCBI) API for requesting the necessary textual data. Subsequently, the operations of data processing and causal analysis will be discussed.

PubMed scraping

The first step of the project consisted in collecting the necessary textual data for testing the causal discovery capabilities of the GPT LLM.

The used data is taken from the PubMed database, a free search engine accessing primarily the MEDLINE database of references and abstracts on life sciences and biomedical topics [Pubmed - wikipedia].

As a first attempt, only the publications’ abstracts (and some extra information) were extracted from the PubMed database.

**Pipeline Operations**

A pipeline handling the essential operations was created for extracting the necessary textual data from the PubMed database. To automate this extraction process, a python script was written using the public API provided by the NCBI as stable interface into its query and database system.

The pipeline allows the user to extract textual data from PubMed by searching for specific terms.

The main operations of the pipeline are the *search\_by\_terms*, *get\_articles\_data*, and the *clean\_data* procedures.

***search\_by\_terms***

The *search\_by\_terms* procedure is the first operation of the pipeline. As the name suggests, it allows the user to search for articles in the PubMed database containing the specified search terms. The search terms are joined as query parameters in the request URL. An API\_KEY is also sent in the request URL, to allow up to 10 requests per second and to ensure smooth and supported access to the desired resources.

def search\_by\_terms(terms, db, retmax, use\_history):

    terms\_string = '+AND+'.join([s.strip().replace(' ', '+') for s in terms])

    url = f'{base\_url}esearch.fcgi?db={db}&term={terms\_string}&retmax={retmax}&api\_key={api\_key}'

    if use\_history:

        url += '&usehistory=y'

    response = requests.get(url)

    ids = re.findall(r"<Id>(\d+)</Id>", response.text)

    if use\_history:

        web\_match = re.search(r"<WebEnv>(\S+)</WebEnv>", response.text)

        web = web\_match.group(1) if web\_match else None

        key\_match = re.search(r"<QueryKey>(\d+)</QueryKey>", response.text)

        key = key\_match.group(1) if key\_match else None

        return ids, web, key

    return ids

The response is in a xml format, and it is processed to extract all ID numbers of the articles found in the specified NCBI database, which in this case is PubMed.

This simplified version of the first pipeline operation only returns the extracted IDs. However, the script allows users to utilize the NCBI *Entrez History* feature, which proves to be significantly more efficient when dealing with tasks that involve searching for or downloading a substantial number of records. This approach helps streamline the process and optimize the retrieval of records in a more efficient manner, making it possible to upload many IDs or download several hundred records at once.

***get\_articles\_data***

The *get\_articles\_data* procedure is the second step of the data acquisition pipeline. It queries the NCBI for the actual content of the articles with the specified ID.

def get\_articles\_data(ids=[], web\_env='', query\_key='', db='pubmed', retmax=1000):

    url = f'{base\_url}efetch.fcgi?db={db}'

    if use\_web\_env:

        url += f'&query\_key={query\_key}&WebEnv={web\_env}'

    else:

        ids\_string = [str(id) for id in ids]

        url += '&id=' + ','.join(ids\_string)

    url += f'&rettype=abstract&retmode=xml&api\_key={api\_key}&retmax={retmax}'

    response = requests.get(url)

    soup = BeautifulSoup(response.text, features="xml")

    articles = soup.find\_all('PubmedArticle')

    if not articles:

        print('ERROR: No articles found')

        return None

    data = pd.DataFrame(columns=['id', 'title', 'abstract', 'keywords', 'pub\_date'])

    for article in articles:

        id = article.find('PMID').get\_text()

        date = article.find('PubMedPubDate', {'PubStatus': 'received'})

        pub\_date = datetime.strptime(f'{date.find("Day").get\_text()} {date.find("Month").get\_text()} {date.find("Year").get\_text()}', "%d %m %Y")

        title = article.find('ArticleTitle').get\_text()

        abstract = ''.join([a.get\_text() for a in article.find\_all('AbstractText')])

        keywords = [k.get\_text() for k in article.find\_all('Keyword')]

        data = pd.concat([data, pd.DataFrame({'id': id, 'title': title, 'abstract': abstract, 'keywords': [keywords], 'pub\_date': pub\_date})]).reset\_index(drop=True)

    return data

Just like before, the shown function is a simplified version of the second step of the pipeline: the NCBI API allows users to query article data with the article ID or by using the *Entrez History* feature, which can provide a more efficient data retrieval. A URL parameter of the request defines the main data content requested, which, in this case, are the abstracts (rettype=abstract).

The returned data is in a xml format, and it is processed and parsed to extract the necessary information. The recovered data include the abstract of the article and some additional information about the article itself. The additional information are the article ID number, the title, the keywords, and the publication date.

***clean\_data***

The *clean\_data* procedure is the third and last step of the data acquisition pipeline. It performs cleaning operations on the obtained data, e.g., by removing null abstract values, duplicates, and eventually removing data of articles published in a particular date range.

def clean\_data(data, drop\_id\_duplicates, drop\_empty\_abstracts, drop\_nan\_abstracts, drop\_date\_nan, drop\_date\_before, drop\_date\_after, search\_terms):

    if data is None or data.empty:

        print('ERROR: No data provided')

        return None

    if drop\_id\_duplicates:

        data = data.drop\_duplicates(subset=['id']).reset\_index(drop=True)

    if drop\_empty\_abstracts:

        data = data.loc[data['abstract'] != ''].reset\_index(drop=True)

    if drop\_nan\_abstracts:

        data = data.dropna(subset=['abstract']).reset\_index(drop=True)

    if drop\_date\_nan:

        data = data.dropna(subset=['pub\_date']).reset\_index(drop=True)

    if drop\_date\_before:

        data = data.loc[data['pub\_date'] > drop\_date\_before].reset\_index(drop=True)

    if drop\_date\_after:

        data = data.loc[data['pub\_date'] < drop\_date\_after].reset\_index(drop=True)

    if search\_terms:

        data['search\_terms'] = [search\_terms]\*len(data)

    return data

GPT interaction

After the preliminary phase of data collection, the actual operations of analysis planned (main focus of the project) for the project were carried out, namely, to investigate the causal capabilities of LLMs, particularly causal discovery. This causal analysis operation consists of learning the causal graphs from a given dataset and context, by uncovering the cause-and-effect relationships and dependencies between the variables and entities of the system of interest: this is done by answering questions such as "Which variables directly affect each other?" or "What is the causal directionality between variables?".

The next step of the project consisted in working with the collected data from the PubMed database to extract information from the abstracts. The necessary information consisted of the main named entities of the textual data, which were subsequently used to perform the actual causal analysis.

This second part of the project was implemented in a python script, in the form of a single pipeline of multiple sub-steps, called *causal\_discovery\_pipeline*: these operations include extracting entities from the textual data, performing the actual causal analysis on the found entities, and ultimately generate the resulting causal graph.

This second part of the project was completely implemented using the GPT API.

**GPT API** [2]

The GPT API is a tool that provides access to OpenAI's GPT models, allowing the integration of natural language processing capabilities into applications. It works by sending requests to the API endpoint with a given prompt, and in return, it generates quality, context-aware text based on the provided input.

The GPT API is used by sending HTTP requests to the API endpoint, specifying the model to use, and providing the necessary parameters such as the prompt and optional additional messages to contextualize the use and behavior of the model, i.e. how the model should answer to requests. The API then processes the request and returns the generated text as a response.

The GPT API can be employed in a variety of applications and use cases. It can be used to generate conversational agents, draft emails or other pieces of writing, provide language translation, answer questions, or assist with content creation.

**Using the GPT API**

The GTP API is used by specifying the model to use (e.g., gpt-3.5-turbo) and additional messages, such as the *system* message and the *user* message. These act as instructions to the model, with the former being a system level instruction to guide the model's behavior throughout the conversation (e.g., asking the model to answer or to act in a specific way), and the latter functioning as the actual request the model is required to answer.

In particular, the system message is used to contextualize the model and its behavior, to make it more useful and accurate for the required operation: for the project’s causal analysis tasks, for example, the prompt was prepended with the message “You are a helpful assistant for causal reasoning”, to try steering the output space to more causally consistent answers. This was shown being an effective prompt-engineering technique that result in more accurate answers [1].

The system message helps set the behavior of the assistant, by modifying the personality of the assistant or providing specific instructions about how it should behave throughout the conversation. However, the system message is optional and the model’s behavior without a system message is likely to be like using a generic message such as "You are a helpful assistant." [2].

messages=[{"role": "system", "content": system\_msg},

          {"role": "user", "content": user\_msg}])

The following is a complete example of a GPT API chat completion request that uses the *gpt-3.5-turbo* model, and specifies system, assistant, and user messages.

openai.ChatCompletion.create(

  model="gpt-3.5-turbo",

  messages=[

        {"role": "system", "content": "You are a helpful assistant."},

        {"role": "user", "content": "Who won the world series in 2020?"},

        {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."},

        {"role": "user", "content": "Where was it played?"}

    ]

)

An example chat completions API response looks as follows:

{

  "choices": [

    {

      "finish\_reason": "stop",

      "index": 0,

      "message": {

        "content": "The 2020 World Series was played in Texas at Globe Life Field in Arlington.",

        "role": "assistant"

      }

    }

  ],

  "created": 1677664795,

  "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",

  "model": "gpt-3.5-turbo-0613",

  "object": "chat.completion",

  "usage": {

    "completion\_tokens": 17,

    "prompt\_tokens": 57,

    "total\_tokens": 74

  }

}

In Python, the assistant’s reply can be extracted as follows:

response['choices'][0]['message']['content']

**GPT Prompt Engineering** [3][4]

Depending on the task, the GPT LLM can produce satisfactory answers when asked to answer a question. However, there are some expedients that have shown to be beneficial and to increase the results accuracy when querying the LLM. These techniques are part of the discipline of prompt-engineering, a set of rules and instructions used to improve the capacity of LLMs on a wide range of common and complex tasks.

Prompt engineering has emerged as a powerful technique to enhance the performance and control the behavior of Language Models (LMs), particularly Large Language Models (LLMs) such as the used GPT-3.5. Prompt engineering involves crafting system and user messages that guide the model's responses and shape its output according to specific requirements.

The goal of prompt engineering is to provide contextual cues and instructions to the language model, enabling it to generate more accurate, relevant, and desired responses. By designing prompts, it is possible to tailor the behavior of LLMs, making them more suitable for various tasks, domains, and user needs.

Among the many prompt engineering techniques, the main ones applied were prepending the prompt with the message “You are a helpful assistant for {task}” (e.g. for the causal discovery task, the system message was “You are a helpful assistant for causal reasoning”) to try steering the output space to more context consistent answers, asking a single question (e.g., regarding the direction of the causal dependency: whether A 🡪 B or A 🡨 B), to answer with a step-by-step explanation [1], and to ultimately give the final answer in an easily parsable way (e.g., in the “<Answer></Answer>” tags).

**NER: Extracting Medical Entities from Text**

As previously mentioned, the second part of the project consisted in working with the collected data. The first step of the operation involved performing Named Entity Recognition on the abstracts, a fundamental procedure to extract and classify named entities. This step was essential for further processing and analysis.

Named Entity Recognition (NER) is a crucial natural language processing (NLP) task that aims to identify and classify named entities within text. In the context of medical texts, NER plays a vital role in extracting specific medical entities such as diseases, symptoms, treatments, drugs, anatomical terms, and medical procedures. Medical texts pose challenges for NER due to their specialized terminology, which often includes abbreviations and multiple names referring to the same concept. Additionally, the complex language structures found in medical texts, along with the diverse sources from which they originate, further complicate the NER process.

The NER operation was performed using the GPT LLM.

**NER gpt prompt messages**

To enhance the performance of the Language Model (LLM) for the NER task, both the system and user messages were designed accordingly.

The system message employed was "You are a helpful assistant for medical Named Entity Recognition" to provide guidance to the model and improve its understanding of the task at hand.

To further aid the model's comprehension, the user message was crafted using the abstract of the medical text, complemented with additional information about the types of entities to be extracted. In this case, since the texts were focused on medical literature and research publications, the model was explicitly instructed to identify entities, with a particular emphasis on "diseases, medications, treatments, and symptoms".

The intention of customizing the user message by providing relevant context and specific entity requirements, was to guide the LLM towards producing more accurate and relevant results for the ongoing NER operation.

def gpt\_ner(text):

    system\_msg = 'You are a helpful assistant for medical Named Entity Recognition'

    user\_msg = f'Given the following text, please identify the named entities, especially diseases, medications, treatments, symptoms. <Text>{text}</Text>. Answer within the tags <Answer><Entity>...</Entity</Answer>.'

    response = gpt\_request(system\_msg, user\_msg)

    if not response:

        return []

    answer\_text = response.choices[0].message.content

    soup = BeautifulSoup(answer\_text, 'xml')

    entities = [entity.text for entity in soup.find\_all('Entity')]

    return entities

The result of the *gpt\_ner* function is an array containing all the found entities; it is then used for the subsequent causal analysis.

The *causal\_discovery\_pipeline* also allows users to add an optional step for entity optimization: by using the GPT api, the pipeline focuses on “*removing redundant or not particularly useful entities that are not diseases, medications, treatments, or symptoms (e.g., “lung cancer” is a valid entity, "lungs" is not).*”

**Causal discovery**

With the NER operation completed and the entities extracted begins the main step of the pipeline, which is the causal discovery operation.

Main description of how function works

Questions asked (edge direction: A -> B / A -> B / A <-> B / A B)

def gpt\_causal\_discovery(entities, text, use\_pretrained\_knowledge, reverse\_variable\_check):

    graph\_edges = []

    system\_msg = 'You are a helpful assistant for causal reasoning'

    text\_msg = ''

    text\_msg += ''

    if text:

        text\_msg += f'the following medical text <Text>{text}</Text> '

        if use\_pretrained\_knowledge:

            text\_msg += 'and '

        else:

            text\_msg += ', '

    if use\_pretrained\_knowledge:

        text\_msg += 'your pre-trained knowledge, '

    for i1, e1 in enumerate(entities):

        for i2, e2 in enumerate(entities):

            if i1 == i2:

                continue

            if not reverse\_variable\_check and i1 >= i2:

                continue

            user\_msg = f'Given {text\_msg}the entities "{e1}" and "{e2}", Which cause-and-effect relationship is more likely? A. "{e1}" causes "{e2}"; B. "{e2}" causes "{e1}"; C: "{e1}" and "{e2}" are not causally related; D: there is a common factor that is the cause for both "{e1}" and "{e2}";  Lets work this out in a step by step way to be sure that we have the right answer. Then provide your final answer within the tags <Answer>[answer]</Answer>, (e.g. <Answer>C</Answer>).'

            response = gpt\_request(system\_msg, user\_msg)

            if response:

                graph\_edges.append(((e1, e2), response.choices[0].message.content))

    return graph\_edges

Sources:

[1] Causal Reasoning and Large Language Models: Opening a New Frontier for Causality, 02/06 (published May, 2023)

[2] [GPT - OpenAI API](https://platform.openai.com/docs/guides/gpt), 29/06

[3] [Prompt Engineering Guide | Prompt Engineering Guide (promptingguide.ai)](https://www.promptingguide.ai/), 30/06

[4] [Investigating causal understanding in LLMs](https://www.lesswrong.com/posts/yZb5eFvDoaqB337X5/investigating-causal-understanding-in-llms), 02/06