Presentation2Article

Elton Cardoso do Nascimento - 233840

IA382 - Seminar in Computer Engineering - 1s2024

The project idea is to use a LLM to generate a informative article draft about the seminar "Talking to Machines: A Practical introduction to Generative AI", by Gabriela Surita, part of 1S/2024 graduate seminar class "IA382" from Unicamp. The project input is the annotations about the seminar, the intended audience and the seminar transcription. The output is the article text draft for subsequent review and editorial treatment, divided in sections. For each section layout information is created, like sentences to highlight and terms to define in auxiliary text boxes. Images will also are proposed for each section (descriptions only, without the real image).

This report, with the used and generated data, can be found in the project Github repository. The original view of this report as a notebook may be more enjoyable to read.

The first step is import the libraries that will be used in this project:

In [ ]:
```python
from __future__ import annotations # Type hints with |

import os #Path operations
import json #JSON read-write
import warnings #Warnings
from typing import Dict, Tuple #Type hints

import torch #spacy don't load if not import before (??)
import spacy #Sentence separator
import tqdm as tqdm #Progress bar
from pyserini.search.lucene import LuceneSearcher #Document search
import sentence_transformers #Reranking
import toolpy as tp #LLM Tools
from toolpy.tool.tool import TextLike #Type hints
from toolpy.integrations import groq #Groq interface
```

The used LLM is the Llama3 70B model, accessed using the Groq API. For creating tools with the model, toolpy is used.

For that, we need to set the model as the default model for the tools:

In [ ]:
```python
groq_interface = groq.GroqInterface(model=groq.GroqModel.LLAMA3_70B, n_retry=5)

registry = tp.llm.LLMRegistry()
registry.registry(model_name="llama3-70b", interface=groq_interface, default=True)
```

Is important to note that the Groq API key must be setted in the environment variable `GROQ_API_KEY` .

Outline

Now that the model is configured, is time for generating a outline of the article. The first tool will use the article theme and intended audience for that.

A short introdution to `toolpy` tools: each tool executes a operation, receiving a query and returning the result. For a complete tool it needs to specify a description of it's operation and it's inputs, and returning the result and a description of it's parts. Several tools are based on performing simple inference with an LLM and are called "BasicTool". They directly define which prompts such as system and user roles will be used, with the operation being performed by the base class, and the return description must be fixed.

Our first tool, "OutlineGenerator" works this way. Observe that, as all the other BasicTools created in this project, they expect the model to return it's results in a JSON format. The BaseClass also parses the LLM inference.

In [ ]:
```python
class OutlineGenerator(tp.BasicTool):
    _description = "Creates a informative text ouline."

    _system_message = '''You are a informative article outline generator that outputs in JSON.
The JSON object must use the schema: {'article_name':'str', 'article':[{'section_name':'str', 'description':'str'}, {'section_name':'str', 'description':'str'}, ...]},

Please use a valid JSON format.'''

    _base_prompt = '''Generate a draft of article sections for the instructions:

Theme: {theme}
Audience: {audience}
'''

    _return_description = {'article_name':"name of the article",
        "article": "list of article sections, with names and descriptions"}

    _input_description = {"theme":"theme of the article", "audience":"intended audience for the article"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```python
outline_generator = OutlineGenerator()
```

BasicTools can also receive a context that is added to the user prompt, before the tool-defined prompt. We gonna use that for injecting personal keypoints annoted during the seminar, for guiding the model:

In [ ]:
```python
context ='''
Personal keypoints:
- Generative AI is better defined as a negative classification: AI that is not for classificattion, regression or control.
- Generative AI generates digital content.
- Examples of gen AI includes sequence modeling (transformers), GAN, VAE, diffusion models.
- The steps for making a query with a text model are: formatting, tokenization, sequence modeling (inference), detokenization and parsing.
- Some model interfaces creates a conversation sequence with turns and roles.
- The majority of models are instruction tuned for something, with techinics like finetuning and reinforcement learning from human feedback.
- Instruction tuning is like a editorial decision from the model creators.
- There is ideological discousers around generative AI, like about emergentism, escaling laws, Turing test, antropomorphism and Shannon divide.
- We should be carefull about metaphors around generative AI, like they can "understand", "think", "reasonate", "halicinate". They can complicate or mislead the models analysis.
- Shannon stated that semantic is irrelevant for the engineering problem of language models. Only the previous sentence words are important to describe the next word probability.
- Shannon divide: is semantic irrelevant? Is dificult to reintroduce semantic, and we would need a "theory of semantics" for that.
- Emergentism: the phenomenon is not described as the sum of parts. It can be a mirage and there is a not undestanded incremental performance improvement.
- Escaling laws: greater the model, greater the performance. But it may not be getting better at what matter for the users (crossentropy is not language habilities).
- Turing test: if mislead a human, it must be inteligent. But what about non-human intelligence capabilities?
- Antropomorphism: makes us let the guard down about what this models and the companies behind they does. Is a design choice.
- Tips for gen AI usage includes: few shot, instruction tuning (not the first choice, is expensive), don't train from the 0 (is really expensive), be aware of the bias and of the hype.

The personal keypoints guide the most important parts of the seminar that will base the article.

The informative article must use the above provided information.
'''
```

The query is defined with the theme and intended audience, as specified by the tool:

In [ ]:
```python
query = {"theme":"generative AI", "audience":"general public"}
```

And we can the generate the article outline and inspect the generated outline:

In [ ]:
```python
outline_result, _ = outline_generator(query, context)
```

In [ ]:
```python
print("Title -", outline_result["article_name"]+"\n")

for section in outline_result["article"]:
    print(f"{section['section_name']}: {section['description']}")
    print("")
```

```
Title - Demystifying Generative AI: Understanding the Concepts and Concerns

Introduction to Generative AI: Defining generative AI as a negative classification: AI that is not for classification, regression, or control, and its ability to generate digital content.

Understanding Generative AI Models: Explaining sequence modeling (transformers), GAN, VAE, diffusion models as examples of generative AI and how they work, including the steps of formatting, token
ization, sequence modeling (inference), detokenization, and parsing.

The Importance of Instruction Tuning: Discussing how most models are instruction tuned for specific tasks using techniques like finetuning and reinforcement learning from human feedback, and how i
t's like an editorial decision from the model creators.

The Ideological Discourse Around Generative AI: Exploring the debates around emergentism, escalating laws, Turing test, anthropomorphism, and Shannon divide, and the need to be cautious with metap
hors that can mislead the analysis of models.

The Limitations of Generative AI: Discussing the limitations of generative AI, including the difficulty of reintroducing semantics and the importance of understanding the capabilities of these mod
els.

Best Practices for Using Generative AI: Providing tips for using generative AI, including few-shot learning, instruction tuning, being aware of bias and hype, and not training from scratch.

Conclusion: The Future of Prompt Engineering: Summarizing the importance of prompt engineering in generative AI and its potential applications for the general public.
```

Reference information

Before the section text generation, we gonna generate a reference information for each section. This reference information will be generated from the transcription of the seminar, using a RAG (Retrieval-Augmented Generation) system.

The first step is to get the seminar transcription and format it:

In [ ]:
```python
with open("data\\Seminar transcription - Only presentation.txt", "r") as file:
    seminar_transcription = file.readlines()

seminar_transcription_filtered = []

for line in seminar_transcription:
    if line == "" or line == "\n":
        continue

    seminar_transcription_filtered.append(line)

seminar_transcription = " ".join(seminar_transcription_filtered)
seminar_transcription = seminar_transcription.replace("\n", "")
```

We then transform it in a set of segments, each segments with `max_length` sentences:

In [ ]:
```python
nlp = spacy.blank("en")
nlp.add_pipe("sentencizer")

#12345
#   45678
stride = 3#2
max_length = 5#3

def window(documents, stride, max_length):
    treated_documents = []

    for j,document in enumerate(tqdm.tqdm(documents)):
        doc = nlp(document)
        sentences = [sent.text.strip() for sent in doc.sents]

        for i in range(0, len(sentences), stride):
            segment = ' '.join(sentences[i:i + max_length])

            treated_documents.append({"contents": segment})

            if i + max_length >= len(sentences):
                break

    return treated_documents

documents = [seminar_transcription]

treated_documents = window(documents, stride, max_length)
```

```
100%|██████████| 1/1 [00:00<00:00, 16.34it/s]
```

In total, we got 119 segments:

In [ ]:
```python
len(treated_documents)
```

Out[ ]:
```
119
```

The segments are exported to a JSONL file:

In [ ]:
```python
if not os.path.isdir("data\iirc_indices"):
    !mkdir data\iirc_indices

file = open("data/iirc_indices/contents.jsonl",'w')

for i, doc in enumerate(treated_documents):
    doc['id'] = i
    if doc['contents'] != "":
        file.write(json.dumps(doc)+"\n")
```

And pyserini is used to create a index from the segments:

In [ ]:
```
!python -m pyserini.index -collection JsonCollection -generator DefaultLuceneDocumentGenerator -threads 1 -input data/iirc_indices -index data/iirc_index -storeRaw
```

```
pyserini.index is deprecated, please use pyserini.index.lucene.
WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will impact performance.
2024-06-25 20:30:48,590 INFO  [main] index.IndexCollection (IndexCollection.java:380) - Setting log level to INFO
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:383) - Starting indexer...
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:384) - ============ Loading Parameters ============
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:385) - DocumentCollection path: data/iirc_indices
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:386) - CollectionClass: JsonCollection
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:387) - Generator: DefaultLuceneDocumentGenerator
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:388) - Threads: 1
2024-06-25 20:30:48,591 INFO  [main] index.IndexCollection (IndexCollection.java:389) - Language: en
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:390) - Stemmer: porter
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:391) - Keep stopwords? false
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:392) - Stopwords: null
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:393) - Store positions? false
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:394) - Store docvectors? false
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:395) - Store document "contents" field? false
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:396) - Store document "raw" field? true
2024-06-25 20:30:48,592 INFO  [main] index.IndexCollection (IndexCollection.java:397) - Additional fields to index: []
2024-06-25 20:30:48,593 INFO  [main] index.IndexCollection (IndexCollection.java:398) - Optimize (merge segments)? false
2024-06-25 20:30:48,593 INFO  [main] index.IndexCollection (IndexCollection.java:399) - Whitelist: null
2024-06-25 20:30:48,593 INFO  [main] index.IndexCollection (IndexCollection.java:400) - Pretokenized?: false
2024-06-25 20:30:48,593 INFO  [main] index.IndexCollection (IndexCollection.java:401) - Index path: data/iirc_index
2024-06-25 20:30:48,595 INFO  [main] index.IndexCollection (IndexCollection.java:481) - ============ Indexing Collection ============
2024-06-25 20:30:48,603 INFO  [main] index.IndexCollection (IndexCollection.java:468) - Using DefaultEnglishAnalyzer
2024-06-25 20:30:48,603 INFO  [main] index.IndexCollection (IndexCollection.java:469) - Stemmer: porter
2024-06-25 20:30:48,603 INFO  [main] index.IndexCollection (IndexCollection.java:470) - Keep stopwords? false
2024-06-25 20:30:48,603 INFO  [main] index.IndexCollection (IndexCollection.java:471) - Stopwords file: null
2024-06-25 20:30:48,695 INFO  [main] index.IndexCollection (IndexCollection.java:510) - Thread pool with 1 threads initialized.
2024-06-25 20:30:48,695 INFO  [main] index.IndexCollection (IndexCollection.java:512) - Initializing collection in data\iirc_indices
2024-06-25 20:30:48,696 INFO  [main] index.IndexCollection (IndexCollection.java:521) - 1 file found
2024-06-25 20:30:48,696 INFO  [main] index.IndexCollection (IndexCollection.java:522) - Starting to index...
2024-06-25 20:30:48,811 ERROR [pool-2-thread-1] index.IndexCollection$LocalIndexerThread (IndexCollection.java:348) - pool-2-thread-1: Unexpected Exception:
java.lang.RuntimeException: Unrecognized token 'gs': was expecting (JSON String, Number, Array, Object or token 'null', 'true' or 'false')
```

```
 at [Source: (BufferedReader); line: 108, column: 3]
        at com.fasterxml.jackson.databind.MappingIterator._handleIOException(MappingIterator.java:420) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.databind.MappingIterator.hasNext(MappingIterator.java:190) ~[anserini-0.22.1-fatjar.jar:?]
        at io.anserini.collection.JsonCollection$Segment.readNext(JsonCollection.java:145) ~[anserini-0.22.1-fatjar.jar:?]
        at io.anserini.collection.FileSegment$1.hasNext(FileSegment.java:136) ~[anserini-0.22.1-fatjar.jar:?]
        at io.anserini.index.IndexCollection$LocalIndexerThread.run(IndexCollection.java:287) [anserini-0.22.1-fatjar.jar:?]
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128) [?:?]
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628) [?:?]
        at java.lang.Thread.run(Thread.java:830) [?:?]
Caused by: com.fasterxml.jackson.core.JsonParseException: Unrecognized token 'gs': was expecting (JSON String, Number, Array, Object or token 'null', 'true' or 'false')
 at [Source: (BufferedReader); line: 108, column: 3]
        at com.fasterxml.jackson.core.JsonParser._constructError(JsonParser.java:2418) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.core.base.ParserMinimalBase._reportError(ParserMinimalBase.java:759) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.core.json.ReaderBasedJsonParser._reportInvalidToken(ReaderBasedJsonParser.java:3038) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.core.json.ReaderBasedJsonParser._handleOddValue(ReaderBasedJsonParser.java:2079) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.core.json.ReaderBasedJsonParser.nextToken(ReaderBasedJsonParser.java:805) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.databind.MappingIterator.hasNextValue(MappingIterator.java:246) ~[anserini-0.22.1-fatjar.jar:?]
        at com.fasterxml.jackson.databind.MappingIterator.hasNext(MappingIterator.java:186) ~[anserini-0.22.1-fatjar.jar:?]
        ... 6 more
2024-06-25 20:30:48,928 WARN  [main] index.IndexCollection (IndexCollection.java:575) - Unexpected difference between number of indexed documents and index maxDoc.
2024-06-25 20:30:48,928 INFO  [main] index.IndexCollection (IndexCollection.java:578) - Indexing Complete! 106 documents indexed
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:579) - ============ Final Counter Values ============
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:580) - indexed:                0
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:581) - unindexable:            0
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:582) - empty:                  0
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:583) - skipped:                0
2024-06-25 20:30:48,929 INFO  [main] index.IndexCollection (IndexCollection.java:584) - errors:                 0
2024-06-25 20:30:48,933 INFO  [main] index.IndexCollection (IndexCollection.java:587) - Total 106 documents indexed in 00:00:00
```

In [ ]:
```python
index_path = "./data/iirc_index"
```

The next tool, "SearchTool", searchs the documents for term correspondences. After that, a dense reranking is used, that is, a cossine similarity between the segments embeddings and query term embedding. The reranking filters the top-k relevant documents.

In [ ]:
```python
class SearchTool(tp.Tool):
    _description = "Searchs for documents with possible usefull information using the query."
    _input_description = {"search_term":"term to search"}
    _return_description = {"search_result":"informating resulted from the search"}


    def __init__(self, index_path:str, embedder_model:str="all-MiniLM-L6-v2", search_k:int=20, rerank_k:int=5) -> None:
        super().__init__(self._description, self._input_description)

        if search_k < rerank_k:
            warnings.warn(f"search_k is less than rerank_k. The result will be of search_k size. ({search_k} < {rerank_k})")

        self._search_k = search_k
        self._rerank_k = rerank_k

        self._embedder = sentence_transformers.SentenceTransformer(embedder_model)
        self._searcher = LuceneSearcher(index_path)

    def _execute(self, query: Dict[str, str] | None, context: str | None=None) -> Tuple[Dict[str, TextLike], Dict[str, str]]:
        #Initial search
        search_term = query["search_term"]
        search_result = self._searcher.search(search_term, k=self._search_k)

        #Get documents
        search_docs = []
        for result in search_result:
            result = json.loads(result.raw)

            search_docs.append(result["contents"])

        #Rerank and filter
        query_embedding = self._embedder.encode(search_term, convert_to_tensor=True)
        search_embeddings = self._embedder.encode(search_docs, convert_to_tensor=True)

        rerank_result = sentence_transformers.util.semantic_search(query_embedding, search_embeddings, top_k=self._rerank_k)

        #Generate response with the selected documents
        response = ""
        for result in rerank_result[0]:
            index = result["corpus_id"]
            result_doc = search_docs[index]

            response += result_doc + "\n"

        response = {"search_result":response}
        return response, self._return_description
```

In [ ]:
```python
search_tool = SearchTool(index_path)
```

A example of the tool usage:

In [ ]:
```python
query = {"search_term":"Shannon"}
result, _ = search_tool(query)
print(result["search_result"])
```

```
So this is the key idea no mathematics so far, but you can ask like, how do you do this? And but those familiar with the answer like for those familiar with  the field. we usually do it with a pro
babilistic model. I tend to think that's specially for language. This is not this is not an obvious leap like and I think it's probably one of  the most  powerful ideas of information Theory and i
nformation transmitting in general like I think Shannon made this leap of faith that if you can.
I tend to think that's specially for language. This is not this is not an obvious leap like and I think it's probably one of  the most  powerful ideas of information Theory and information transmi
tting in general like I think Shannon made this leap of faith that if you can. rap sentences out of their meaning and just model them as  probabilities you can actually do something some. like qui
te powerful things like and basically the the key idea behind this is that you can express. the probability of the next element in a sequence for example the probability of fox given the quick Bro
wn as You can express this as probability distributions over the next word and you can probably maximize probability of fox given to a quick brown and you can maximize probability of 13 given the
beginning of the supernet.
So be aware of entrepromorphizing and I think the final thing that I want to talk about is the shenon Divide. I think this is a term that I  invite invented but if you go back to this light on Sha
nnon, I think like by construction We removed any notion of meaning from symbols and we built our whole system around this  property that so by construction we removed meaning so it's hard to rein
troduce meaning like when you take the outputs of assistance without a theory a theory of semantics. So this doesn't say that these series  of semantics don't exist, but you need one like otherwis
e, it's there is a There's a flaw in your like. epistemic step by step. reasoning on how how many arises so I think the Practical conclusion here is  be very careful when you apply meaning and thi
s usually showing metaphors to the outputs of these systems.
You can also measure like probabilities probably the probabilities of bananas given the experiment suggests is very unlikely. but I think this idea of expressing. sequence the next element of a se
quence as a probability over all possible elements vocabulary and given the conditionals of the previous elements of the sequence a very very powerful idea dear and I think like is a direct result
from Shannon's theory of communication. So this is quite important. We're gonna go back to this sometime soon.
There's also does notification like it's making a look like Disney characters. I think that's that's also phenomena that happens. Yeah. So be aware of entrepromorphizing and I think the final thin
g that I want to talk about is the shenon Divide. I think this is a term that I  invite invented but if you go back to this light on Shannon, I think like by construction We removed any notion of
meaning from symbols and we built our whole system around this  property that so by construction we removed meaning so it's hard to reintroduce meaning like when you take the outputs of assistance
without a theory a theory of semantics.
```

But what to search for? The next tools selects a list of serach terms from a topic and it's description:

In [ ]:
```python
class SearchTermSelector(tp.BasicTool):
    _description = "Selects a list of terms to search for a better topic undestanding."

    _system_message = '''You are a search term selector that outputs in JSON.
The JSON object must use the schema: {'search_terms':['str', 'str', ...]},

Please use a valid JSON format.'''

    _base_prompt = '''Select terms for a better undestanding of the topic:

Topic: {topic}
Topic description: {description}
'''

    _return_description = {"search_terms": "list of search terms"}

    _input_description = {"topic":"topic to search form", "description":"brief topic description for directing the search"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
search_term_selector = SearchTermSelector()
```

We can then generate search terms from each section, using its name as the topic and its description as the topic description:

In [ ]:
```
for section in outline_result["article"]:
    query = {"topic":section["section_name"], "description":section["description"]}

    terms_result, _ = search_term_selector(query)

    section["search_terms"] = terms_result["search_terms"]
```

In [ ]:
```
outline_result["article"][0]["search_terms"]
```

Out[ ]:
```
['generative ai definition',
 'ai types',
 'machine learning categories',
 'digital content creation',
 'non classification ai',
 'ai versus ml',
 'deep learning applications',
 'creative ai',
 'intelligent content generation']
```

However, the pure search result will be too much text for the Groq API token limit, and it can also confuse the model, being a very long text with repeated information. The Summarizer tool is created for summarizing a text according to a focal topic:

In [ ]:
```
class Summarizer(tp.BasicTool):
    _description = "Summarizes a text according to a focal topic."

    _system_message = '''You are a text summarizer that outputs in JSON.
The JSON object must use the schema: {'summary':'str'},

Please use a valid JSON format.'''

    _base_prompt = '''{text}

Summarizes the above text, focusing on the following topic:

Focal topic: {topic}
'''

    _return_description = {"summary": "summarized text"}

    _input_description = {"text":"text to summarize", "topic":"topic to focus the summary"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
summarizer = Summarizer()
```

Finally, the section reference information can be generated, searching for the search terms in the seminar transcript, and summarizing the retrived information focusing in the search term:

In [ ]:
```
for section in outline_result["article"]:
    section["reference_information"] = []

    for search_term in section["search_terms"]:
        query = {"search_term":search_term}
        search_result, _ = search_tool(query)

        query = {"text":search_result["search_result"], "topic":search_term}
        summary_result, _ = summarizer(query)

        section["reference_information"].append(summary_result)
```

In [ ]:
```
outline_result["article"][0]["reference_information"]
```

Out[ ]:
```
[{'summary': "Generative AI refers to AI that produces unbounded digital content, often associated with creative tasks, and involves sequence modeling, which is a predictive system that can predic
t the next element of a sequence. It's differentiated from other types of AI, such as predictive AI, and is key to understanding systems that involve producing content."},
 {'summary': 'There are two main types of AI: predictive AI, associated with statistical tasks, and generative AI, associated with open-ended creative tasks and digital content creation. Generativ
e AI includes chatbots, text-to-image systems, and other multimodal agents, and is often differentiated from narrow AI.'},
 {'summary': 'The text discusses machine learning categories, distinguishing them from classification and regression, and introduces the concept of human consumable content generation, such as tex
t, videos, images, and audio, enabled by machines or systems. It highlights that this type of machine learning is different from traditional classification and regression, and is capable of produc
ing unique compilations of content.'},
 {'summary': 'Generative AI is a type of AI that produces human-consumable digital content, such as text, videos, images, and audio, and is not limited to classification, regression, or multitask
learning. Examples of generative AI include chatbots, text-to-image and text-to-audio systems, which can generate new content.'},
 {'summary': 'Generative AI is a type of AI that produces human-consumable content such as text, videos, images, or audio, and is often associated with open-ended creative tasks, differentiating i
t from predictive AI which is associated with statistical tasks. It does not involve classification, regression, or multi-choice action selection, and is exemplified by language models like GPT, G
emini, and Claude, as well as chatbots and text-to-image systems.'},
 {'summary': 'The text does not explicitly discuss the difference between AI and ML. However, it mentions AI subfields such as predictive AI, associated with statistical tasks, and generative AI,
associated with open-ended creative tasks. The speaker highlights the importance of recognizing intelligence that does not look human, citing examples like AIS predicting protein structures and ge
nerating AI as transformative opportunities in the field.'},
 {'summary': 'The speaker discusses deep learning applications, specifically in text completion and chatbots, explaining how they work and how to train large sequence models. The presentation will
cover hands-on tips and tricks for using these models, as well as providing an introduction to generative AI and its applications.'},
 {'summary': "Generative AI is associated with open-ended creative tasks, distinguishes from predictive AI which is classed with statistical tasks. It enables modern AI assistants, such as chatbot
s and text-to-image systems, which are incredibly powerful and can help humans, but may not be recognized as intelligent because they don't look like human intelligence."},
 {'summary': 'The speaker is introducing the concept of generative AI, specifically sequence modeling, which is a key component in modern chatbots and other interactive systems. Sequence models ar
e predictive systems that generate the next element in a sequence, and understanding them is crucial for developing intelligent content generation. The presentation will cover the technical basics
of sequence modeling and its applications in AI, followed by a more abstract discussion on the interpretations and philosophy of AI.'}]
```

Observe that, because of the parallel sequence of operations used according to each search term, the reference information can contain repeated information between terms.

Text generator

With the names, descriptions and reference information of the sections, we can generate their text.

The SectionTextGenerator uses this informations for that:

In [ ]:
```
class SectionTextGenerator(tp.BasicTool):
    _description = "Creates a informative text section."

    _system_message = '''You are a informative article writter that outputs in JSON.
The JSON object MUST use the schema: {'section_text':'str'},

Please use a valid JSON format.'''

    _base_prompt = '''Write the text for the the text for the following section of the text:

Section name: {section_name}
Section description: {section_description}
Text intended audience: {audience}

Don't forget to use the JSON schema: {{'section_text':'str'}}
'''

    _return_description = {"section_text": "section text"}

    _input_description = {"section_name":"name of the section",
                      "section_description":"description of the section",
                      "audience":"text intended audience"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
section_text_generator = SectionTextGenerator()
```

As the first tool, a context is created with the reference information for injecting this information in the tool.

Also, because of the more complex and long prompt, the LLM result can sometimes not be in a valid JSON format. We repeat the query if that occurs.

In [ ]:
```
for section in tqdm.tqdm(outline_result["article"]):
    reference_information = [information["summary"] for information in section["reference_information"]]
    reference_information = "\n".join(reference_information)

    context = '''{reference_information}

Use the information above as a basis when writing the text for the section.'''

    query = {"section_name":section["section_name"],
             "section_description":section["description"],
             "audience":"general public"}

    generator_result = None
    while generator_result is None:
        try:
            generator_result, _ = section_text_generator(query)
        except groq.groq.groq.BadRequestError:
            pass


    section["text"] = generator_result["section_text"]
```

```
100%|██████████| 7/7 [00:08<00:00,  1.15s/it]
```

In [ ]:
```
print(outline_result["article"][0]["text"])
```

```
Generative AI is a type of artificial intelligence that doesn't fit into the traditional categories of AI, such as classification, regression, or control. Instead, its primary function is to creat
e new, original digital content. This can include images, music, text, and even entire videos. Think of it as a creative partner that can generate ideas, complete tasks, and bring new concepts to
life. With its ability to produce novel and diverse content, generative AI is revolutionizing industries and changing the way we experience digital media.
```

Special elements

The next step is to create special elements blocks. Three elements will be generated:

- Highlights: quotes from the text to highlight next to it paragraph.
- Definitions: terms to define.
- Images: proposal of images for the section.

They are editorial elements that will need further processing before the creation of the final article.

Highlights

For the highlights, we create a tool to select sentences from the sections to highlight:

In [ ]:
```
class HighlightedSentencesSelector(tp.BasicTool):
    _description = "Selects sentences to higlight from a scientific text."

    _system_message = '''You are an article editor for a scientific journal. Your role at this point is to select sentences to highlight.
The JSON object MUST use the schema: {'sentences': ['str', 'str', 'str', ...]}, where sentences must be extracted from the article.

Please use a valid JSON format.'''

    _base_prompt = '''Select sentences to highlight from the text.
Note that few, if any, sentences should be highlighted. It must be something key to the section.

Text intended audience: {audience}
Text name: {name}
Text description: {description}
Text : {text}

Don't forget to use the JSON schema: {{'sentences': ['str', 'str', 'str', ...]}}
'''

    _return_description = {"sentences": "list of sentences to highlight"}

    _input_description = {"name":"name of the section",
                      "description":"description of the section",
                      "text":"text of the section.",
                      "audience":"text intended audience"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
highlight_selector = HighlightedSentencesSelector()
```

In [ ]:
```
for section in tqdm.tqdm(outline_result["article"]):

    query = {"name":section["section_name"],
             "description":section["description"],
             "audience":"general public",
             "text":section["text"]}

    highlight_result = None
    while highlight_result is None:
        try:
            highlight_result, _ = highlight_selector(query)
        except groq.groq.groq.BadRequestError:
            pass


    section["higlighted_sentences"] = highlight_result["sentences"]
```

```
100%|██████████| 7/7 [00:15<00:00,  2.21s/it]
```

In [ ]:
```
outline_result["article"][0]["higlighted_sentences"][0]
```

Out[ ]:
```
"Generative AI is a type of artificial intelligence that doesn't fit into the traditional categories of AI, such as classification, regression, or control."
```

But, the LLM can sometimes select sentences that aren't in the text, or that not appears exactly as presented. Because of that we need to filter out incorrect sentences:

In [ ]:
```
for section in outline_result["article"]:

    to_remove = []
    for sentence in section["higlighted_sentences"]:
        if sentence not in section["text"]:
            to_remove.append(sentence)

    for sentence in to_remove:
        section["higlighted_sentences"].remove(sentence)
```

Definitions

The definitions are selected in a similar manner. They are selected terms to define from the text, with the definition, both created using the model:

In [ ]:
```
class DefinitionSelector(tp.BasicTool):
    _description = "Selects terms to define in a scientific text."

    _system_message = '''You are an article editor for a scientific journal. Your role at this point is to select terms that may be unfamiliar to the audience and create a definition to place in a
The JSON object must use the schema: {'terms': [{'term':'str', 'definition':'str'}, {'term':'str', 'definition':'str'}, ...]}, where the 'term' must be extracted from the article.

Please use a valid JSON format.'''

    _base_prompt = '''Select terms to define from the section text.
Note that few, if any, terms should be defined. It must be something key to the section and that the audience may not know.

Text intended audience: {audience}
Text name: {name}
Text description: {description}
Text : {text}

Don't forget to use the specified JSON schema.
'''

    _return_description = {"terms": "list of terms to define, with the term and definition"}

    _input_description = {"name":"name of the section",
                          "description":"description of the section",
                          "text":"text of the section.",
                          "audience":"text intended audience"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
definition_selector = DefinitionSelector()
```

In [ ]:
```
for section in tqdm.tqdm(outline_result["article"]):

    query = {"name":section["section_name"],
             "description":section["description"],
             "audience":"general public",
             "text":section["text"]}

    definition_result = None
    while definition_result is None:
        try:
            definition_result, _ = definition_selector(query)
            section["definitions"] = definition_result["terms"]

        except groq.groq.groq.BadRequestError:
            definition_result = None
            pass
```

```
100%|██████████| 7/7 [00:04<00:00,  1.68it/s]
```

In [ ]:
```
outline_result["article"][0]["definitions"]
```

Out[ ]:
```
[{'term': 'regression',
  'definition': 'A type of machine learning task where a model predicts a continuous or numerical value based on input data.'}]
```

Images

The images are proposed for each section text.

In [ ]:
```
class ImageProposer(tp.BasicTool):
    _description = "Proposes images for a scientific text."

    _system_message = '''You are an article editor for a scientific journal. Your role at this point is to propose images for the text.
The JSON object must use the schema: {'images': [{'name':'str', 'description':'str'}, {'name':'str', 'description':'str'}, ...]}.

Please use a valid JSON format.'''

    _base_prompt = '''Propose images for the text.
Note that few, if any, images should be proposed. It must be something key to the section and usefull for the audience.

Text intended audience: {audience}
Text name: {name}
Text description: {description}
Text : {text}

Don't forget to use the specified JSON schema.
'''

    _return_description = {"images": "list of proposed images, with name and description"}

    _input_description = {"name":"name of the section",
                          "description":"description of the section",
                          "text":"text of the section.",
                          "audience":"text intended audience"}

    def __init__(self, model_name: str | None = None) -> None:
        super().__init__(self._description, self._input_description,
                         self._base_prompt, self._return_description,
                         self._system_message, model_name, True)
```

In [ ]:
```
image_proposer = ImageProposer()
```

In [ ]:
```
for section in tqdm.tqdm(outline_result["article"]):

    query = {"name":section["section_name"],
             "description":section["description"],
             "audience":"general public",
             "text":section["text"]}

    result = None
    while result is None:
        try:
            result, _ = image_proposer(query)
            section["images"] = result["images"]

        except groq.groq.groq.BadRequestError:
            result = None
            pass
```

```
100%|██████████| 7/7 [00:04<00:00,  1.69it/s]
```

In [ ]:

```
outline_result["article"][0]["images"]
```

Out[ ]:

```
[{'name': 'Generative AI Creative Partner',
  'description': 'Illustration of a robot partner with a creative spark, surrounded by digital content'}]
```

Article export

With the generated informations, we than export a draft. We will use a Markdown format for that, in special the "Obsidian" Markdown flavor.

Before, the `generate_callout` function is defined for creating callouts for the special elements:

In [ ]:

```
def generate_callout(text:str, type:str, name:str="") -> str:
    text = "> " + text.replace("\n", "\n> ")
    callout = f"> [!{type}]- {name}\n"+text

    return callout
```

We than export the final draft:

In [ ]:

```
with open("generated_article.md", "w") as file:
    file.write(f"# {outline_result['article_name']}\n")

    abstract = ""
    for section in outline_result["article"]:
        abstract += f"1. **{section['section_name']}**: {section['description']}\n"

    abstract = generate_callout(abstract, "abstract")
    file.write(abstract+"\n\n")

    for section_index, section in enumerate(outline_result["article"]):
        file.write(f"## {section_index+1} - {section['section_name']}\n")

        for image in section["images"]:
            image_callout = generate_callout(image["description"],
                                             "todo",
                                             f"Image - {image['name']}")

            file.write(image_callout+"\n\n")

        paragraphs = section["text"].split("\n")

        for paragraph in paragraphs:
            for sentence in section["higlighted_sentences"]:
                if sentence in paragraph:
                    quotation = generate_callout(sentence, "quote")
                    file.write(quotation+"\n\n")

            for definition in section["definitions"]:
                if definition["term"] in paragraph:
                    definition = generate_callout(definition["definition"],
                                                  "info",
                                                  definition["term"])

                    file.write(definition+"\n\n")

            file.write(paragraph+"\n")
```

The rendered **result** can be viewed in the project repository: generated_article.pdf.

It contains all the generated information, and can be used as a starting point for creating the final article, with suggestions of sections, texts, images, quotes and definitions.

Conclusion

Analyzing the generated draft contents, it is possible to observe how it contains information taken from the presentation, while also including some new information that was existing in the model weights, like the other model types definitions (GAN, VAE). The presented technique can be useful for generating informative materials for presentations, and perhaps even expository classes.

However, a bad aspect of the generated article is the number of sections with just one paragraph. The key topics also seem to have restricted the information used from the seminar too much, which may not be desirable.

Futures possible developments includes:

- Use of information from selected articles during the discipline's bibliographic review task
- Export to a LaTeX file with formatting closer to the final article, facilitating the editorial process.
- Use of a image generative model for generating drafts of the proposed images.
- Perform a global summary of the seminar as an alternative to the information used to generate the outline (key topics).