LLMs (72 points possible)

In this assignment, we'll explore how to augment existing LLM models with capabilities such as

- Referring to custom documents (RAG)
- Calling functions that augment the agent's capabilities (ReAct)
- Adding memory

The main document we'll be using to augment the agent is a *Dungeons and Dragons* adventure - a sketch of a story and some game statistics for the major characters. It's a little weird, since it's set in the decidedly weird *Spelljammer* setting of "D&D in space," but the main reason we're using it is that it's a good example of a custom document an LLM might need to refer to.

```
!pip install langchain_community llama-cpp-python google-genai langchain-google-genai
```

1. We're first going to get a barebones LLM running, and also demonstrate that the context window of a modern LLM tends to be quite large.

a, 3 pts) Look up the size in tokens of the Gemini 3 Flash context window. Compare this to a word count for Spelljammer23.txt (using the wc unix utility, for example), keeping in mind that 1 token is typically 0.75 words. Does it fit?

TODO does it fit?

b, 4 pts) Use the example from lecture to create a ChatGoogleGenerativeAI llm that uses Gemini 3 Flash as a backend. Then write a function ask_about_adventure(query, llm) that queries Gemini 3 Flash with question *query* with the additional context of the Spelljammer23.txt file, returning the string that is the LLM's answer. Do not use a vector store for this; just dump the whole document in the query. (Note that if we use remote calls to Gemini 3 Flash as our LLM, this assignment should be doable even in free Colab.)

```
# TODO create llm, code ask_about_adventure
```

```
ask_about_adventure("What is Kip's class?", llm) # Expect Cleric
```

2, 10 pts) We're now going to pretend this file is actually too big to fit in the context window, to illustrate how to implement RAG.

Code up a RAG-enabled agent with the help of the examples from lecture, so that our my_rag_app test snippets refer to an object like the RAGApplication defined there. The text is still Spelljammer23.txt. Note that the blocks of statistics in the adventure tend to not have periods, so instead of following the lecture example exactly, chunk the text into 1000 character strings that overlap by 100 characters. Use the same Gemini 3 Flash model as the previous LLM. Retrieve the 4 best documents on querying the vector store.

```
# TODO chunk document into documents
```

```
documents[10] # Expect powers of the magic artifact and some nearby treasure
```

```
# TODO create vector store, retriever
```

```
# TODO create my_rag_app RAGApplication
```

```
my_rag_app.run("What is Kip's character class?") # Expect Cleric
```

```
my_rag_app.run("What is Gardia's brother's name?") # Expect Cornelius
```

3, 13 pts) Let's add to the agent the ability to roll dice. Follow the example at https://python.langchain.com/docs/how_to/custom_tools/ to create a function roll(to_roll) that takes a string "*n*ds" (like "2d10"), rolls *n* dice with faces numbered 1 through *s* (like 2 10-sided dice), and returns the total. Be sure to use the tool decorator and give it a text docstring so that the LLM knows what the function is for.

When you have a basic "nds" roller, make it more robust in the following ways, in case the AI tries to pass weird arguments:

a. Have the docstring contain explicit instructions to not pass any additional modifiers or comments besides NdS, where N is the number of dice and S is the number of sides.

b. Use a regular expression to grab the "nds" part of the argument so that if it says something like "1d20+7" or "1d20 (for the attack)" you can ignore the additional text.

c. If the argument is still unparseable, return the message "Badly formatted input - use just NdS with no modifiers or comments."

```
my_rag_app.run("Roll 2d10.") # Expect "I don't know."
```

```
# TODO define roll tool
```

```
print(roll.invoke({'to_roll':'3d4'})) # Produce random number between 3 and 12
print(roll.invoke({'to_roll':'1d20+7 (for stealth check)'})) # random number between 1 an
print(roll.invoke({'to_roll': 'my lucky die'})) # produce error message
```

Now create an agent that can use this die-rolling function when necessary. It doesn't need to have the RAG capabilities for this iteration.

```
# TODO create_agent call
```

```
result = agent.invoke({
    "messages": [
        {"role": "user", "content": "Roll 2d20."}
    ]
})
print(result['messages'][-1].content[0]['text']) # Expect a random result in 2-40
```

```
result = agent.invoke({
    "messages": [
        {"role": "user", "content": "Make a Perception check with +4."}
    ]
})
print(result['messages'][-1].content[0]['text']) # it knows some D&D rules already!
```

4, 13 pts) Let's try putting the previous two steps together. Give your die-rolling agent RAG capabilities, and add new instructions indicating that the agent is now a "Dungeon Master" narrating the game, and the user is playing the hero Gardia. (The Dungeon Master role instruction will go a long way toward getting the agent to behave reasonably how we want.) Create a loop that asks the player what to do next.

Before the player has said anything, use an input of 'Begin the adventure!' and the RAG context should just be the first four chunks of the document.

Note that your invoke call for your agent just wants one string *content* in an argument of the form

`{"messages": [{"role": "user", "content": content}]}`

so if you want to pass relevant documents for RAG, concatenate them to the user input and pass one big string, like

`content = input + '\nRelevant_documents: ' + doc_texts`

Some additional instructions:

- Tell the agent that it should use its roll tool to roll dice to resolve whether player and non-player character actions succeed, and tell it to explicitly announce those die roll outcomes. (We want to be able to see that the die rolling is really happening.) To make sure we can see this happen in testing, instruct it to *always* make at least one roll and announce it.
- Tell it that it should only announce one result of the player's action and any reactions from non-player characters, and it should then ask what the player wants to do and end the response. Tell it to not take multiple turns in the narration.

```
# TODO create the system prompt string, call create_agent
```

```
# TODO adapt RAGApplication code to use the agent
```

Now, this is not going to work fully, as without memory the narration will jump around the plot at the whim of the RAG retrieval. Nevertheless, show a reasonable interaction that lasts for at least 5 player inputs where a die is rolled at least once (sneaking and fighting are good for die rolling).

```
my_input = 'Begin the adventure!'

while(True):
  print(my_dm.run(my_input))
  my_input = input()
```

5, 10 pts) This would be a little better if the agent had memory. Modify the class you used to implement `my_dm` so that it has a memory of the last k utterances (let k be a constructor argument) of either user or AI. You don't need to maintain a special class; just implement a list of utterances that drops the oldest memory when it gets too big. When the RAG-augmented string is being constructed for input, also add a string "chat history:" followed by the transcript you stored. Create a new agent where k = 100.

```
# TODO adapt RAGApplication code to insert last k utterances in prompt
# (you don't need to use a special memory class for this, just maintain it yourself)
```

Try running this new `my_dm` agent for 5 sessions. Your fifth input should be "recap the story so far", which should check whether the agent's memory is working properly. (Some jumping around in the story is still possible if the RAG decides a different part of the story is most relevant to the action.)

```
my_input = 'Begin the adventure!'

while(True):
  print(my_dm.run(my_input))
  my_input = input()
```

6, 13 points) For our last exercise using this file, we'll experiment with chaining LLM outputs. Create a chain that can take as input a document (like Spelljammer23.txt) and output two good titles for that document. But do it by chaining LLMs in LangChain: the first LLM generates a first candidate title, the second generates a second candidate title, and the third rates these according to their originality, how exciting they are, and their faithfulness to the original document, replying with the three scores for each title and the overall winner. (The ratings don't need to be on any particular scale.)

Include instructions in your second prompt to try to create a very different title from the first prompt (use the first title in its prompt).

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

with open('Spelljammer23.txt', 'r') as f:
  document = f.read()
```

```
# TODO prompts and chaining to create llm_chain
```

```
llm_chain.invoke({'document': document})
```

7, 4 pts) Last question: Suppose in my prompt I make use of a word that an LLM has never seen in training, like “probiognosis.” Explain how an LLM with byte-pair encoding (BPE) would react to this word, both for the tokenization and the meanings of these tokens. Compare to what would happen if the LLM instead just tokenized using whitespace and punctuation.

TODO

AI Statement (2 pts)

Please briefly describe whether and how you used generative AI for this assignment. You will not be penalized for your answer - this is mostly so the course can adapt to AI use.

TODO