ConversationalAssistants

September 25, 2024

1 LangChain for Conversational Assistants

In this assignment, you will make use of the LangChain library to connect different tools and LLM outputs together to form a conversational system that will play the role of a personalized, conversational assistant. This assignment will consist of a sequence of steps utilized to create a meaningful LangChain structure:

1. Use an LLM to handle typical conversational assistant needs.
    - Understand if the model needs to access the external file.
    - Summarize different information provided to the model.
    - Converse with the user.
2. Understand the use of components.
    - Understand the use cases of a VectorStore database (i.e. information extraction).
    - Understand the use cases of LLMs with different system prompts.
3. Link systems with LangChain.
4. Gain a greater understanding of the Huggingface Transformers library.
    - Utilize the database to gain more information about NLP.

In an exploration of the bonus material, you will additionally:

1. Understand how to load new tools utilizing the LangChain package.
    - Specifically, understand the use of the ArXiv search tool.

1.1 Write your name here:

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version *ConversationalAssistants.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *ConversationalAssistants.ipynb*.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing we will see when grading!
7. Submit **both** your PDF and notebook on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

Use this cell to install LangChain and the required packages.

```
[ ]: !pip install -q langchain_core
!pip install -q langchain
!pip install -q langchain_community
!pip install -q lancedb
!pip install -q langchain_openai
!pip install -q gdown
!pip install -q arxiv
!pip install -q sentence-transformers
!pip install -q optimum
!pip install -q onnx
```

3 Setting up the VectorStore database

The code below sets up a LanceDB vector database. Utilizing the OpenAIEmbeddings class, it sets up an embedding for the system to utilize to take text and project the information into a table of vectors.

Running this cell is only required for Google Colab, otherwise the file was provided in `hw04/data/lancedb_jina_code`. Since the folders are structured by the output of LanceDB, and large, it is more convenient to use this method to import the data.

```
[ ]: import gdown
gdown.download_folder("https://drive.google.com/drive/folders/
  ↪1qafKNGxn1SrKIVlCfAwAx0T-eIiKsEYL",
      quiet = True,
      output = "../data/lancedb_jina_code")
```

If the cell above does not work, create a new folder called `lancedb_jina` in the same directory as this notebook and then download the folder `hf_docs.lance` from https://drive.google.com/drive/folders/1VnioI8X9ZtzCWqJhu5PG-itLKAHszgkI and extract it into the `lancedb_jina` folder.

4 Huggingface Login

This homework assignment requires some work with Huggingface Embeddings. Because of this, you must make an account with Huggingface. 1. Make an account with Huggingface at https://huggingface.co 2. In the top right, click the profile picture and navigate to `Settings`. 3. Navigate on the left side-bar to `Access Tokens`. 4. Create a token with `Read` access, it can be named whatever you want. Copy this token. 5. Run the code below, provide your Huggingf ace access token. 6. Navigate to https://huggingface.co/jinaai/jina-embeddings-v2-base-code. 7. Accept the terms of use of the embeddings.

After this, you should have access to the use of these word embeddings for our LanceDB vectorstore database.

```
[ ]: from huggingface_hub import notebook_login

notebook_login()
# If you struggle with the above, you can try login(token = access_token) where␣
 ↪access_token is loaded from a
# .env file.
```

```
[ ]: import lancedb
from langchain_lancedb.lc_lancedb import LanceDB
from langchain_community.embeddings import HuggingFaceEmbeddings
from transformers import AutoModel

db = lancedb.connect('../data/lancedb_jina_code')

# Sets up the embeddings to use the embeddings
embedding_fn = HuggingFaceEmbeddings(model_name = "jinaai/
 ↪jina-embeddings-v2-base-code",
                                     model_kwargs = {'trust_remote_code' :␣
 ↪True})
vectorstore = LanceDB(db, embedding_fn, table_name = "hf_docs")
```

5 What is the Huggingface Documentation?

Huggingface's documentation has information about many Natural Language Processing tasks. The database is scraped from the website, along with links to the original webpages, and encoded using the jina-embeddings imported above.

You can use this database to answer many different questions about course material, which is what we will use this database for later in the assignment. For example, the answer to the question in the code below should mention that a word embedding is a vector to describe the meaning of a word.

The vector database is created using a model for word embeddings that was trained on a combination of text and code. These embeddings are the basis of how the vector store database for this assignment operates.

This database also contains embeddings of the Huggingface NLP courses, so it contains a wealth of information about NLP.

Feel free to explore the database as much as you would like, for this homework assignment and as the course continues.

6 1.a. Utilizing LanceDB database to solve queries (5 points)

In the function read_database(query), given a string query, use vectorstore.similarity_search(query), which takes as input a string query and returns a list of Documents. Return the page content of the first Document of the list, accessed via the page_content attribute.

If you are interested in learning more about how LanceDB operates, there is a description

available at https://lancedb.github.io/lancedb/concepts/vector_search/. There are a few graphics there to make it more clear what the database is doing and how it is finding relevant results, given the word embeddings of your query (which are computed automatically by vectorstore.similarity_search(query). There is also some intuition for why we are using vector databases on Slide 11 of the LangChain slides on the course website.

[ ]:
```
def read_database(query):
  ## YOUR CODE HERE

    return ''

# This should return the page content for https://huggingface.co/learn/
 ↪nlp-course/en/chapter1/2
print(read_database("What is a word embedding?"))
```

[ ]:
```
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.prompts import ChatPromptTemplate
from langchain import LLMChain
from langchain.agents import load_tools

# These are set up for the Llama-3 model utilizing the A5000 GPUs
# Feel free to change the information to match the model of your preference.
# For your own security, do NOT input your API key for OpenAI as a string in␣
 ↪this cell.
# Use a .env file, or if you are using Colab, use the Secrets tab.
api_key = "aewndfoa1235123"
base_url = "http://cci-llm.charlotte.edu/api/v1"

model_name = "/quobyte/ealhossa/hf_models/Llama-3-70B"
```

7 1.b. Basic utilization of the LLM with LangChain (10 points)

In the function `respond(messages)` take as input a list of tuples `messages`, and create a ChatPromptTemplate from `messages`. `respond(messages)` should return a string containing the LLM response to the conversation.

This is very similar to the implementation seen in the second example from the LangChain Jupyter Notebook on the course website.

Hint: ChatPromptTemplate has a method from_messages(listMessages) where listMessages is a list of tuples, with the first element being the role and the second element being the content. This can be given as a prompt to LLMChain. LLMChain can be instantiated with named parameters llm and prompt. It can then be executed with invoke(dict()), since we do not have any formatting in our string, we need to pass an empty dictionary. Remember that invoke returns a dictionary, we want only the string response.

[ ]:
```
def respond(messages):
    llm = ChatOpenAI(api_key = api_key, base_url = base_url,
                 model = model, temperature= 0, max_tokens = 1000)

    answer = ''
    ## YOUR CODE HERE
    # Create a ChatPromptTemplate.


    # Create an LLMChain.


    # Get the answer from LLMChain with the invoke() method.
    # Remember: LLMChain returns a dictionary, you want to respond with only␣
 ↪the string output.


    return answer

messages = [
    (
        'user',
        'Tell me what you know about word embeddings.'
    )
]

print(respond(messages))
```

8 2. LLMs as Tools (20 + 15 points)

While LLMs are used as a component that controls what tools are used, one can also have an LLM with a specific steering prompt as a tool in itself. In this case, this LLM tool will be used as a fact checker. It should be given as input a statement about that topic, and call your `query_database(query)` function to get information. The LLM should determine if the statement follows from the information.

This component should return three things: - True or False - An explanation for its answer - The link from the documentation that it referenced.

Hint: LLMs understand Markdown, JSON, and other formatting methods. Utilizing these make it easier for the LLM to understand the difference between different entities in your prompt.

[ ]:
```
## YOUR CODE HERE ##
# Create a system message for an LLM that determines if something is true,
# given information and a statement about that information.
system_message = ""
```

```
# Create instructions containing {information} and {statement} to be used by
# the prompt template.
instructions = ""
```

Additionally, replace the "## YOUR TOOL DESCRIPTION HERE ##" with a description of what the tool does, and complete the code for the fact_check function such that it gets the response of the model, given the user's message and your system message and instructions.

[ ]:
```
from langchain.tools import tool

# This decorator defines a tool with the name fact_check
@tool
def fact_check(statement):
    # Place in this string (do not store it anywhere, this describes the tool␣
 ↪to the model)
    # a description of what this tool does. This will describe to the model
    # when to use this tool.
    '''
    ## YOUR TOOL DESCRIPTION HERE ##
    '''

    ## YOUR CODE HERE ##

    # Create a list of tuples in the form (role, content), where role is the␣
 ↪speaker
    # and content is what is said.
    messages = []

    return respond(messages)
```

9 3. Retrievers as Tools (10 points)

Vector Stores can be used as tools, referred to as Retrievers. These are tools that are utilized to retrieve data from a source external to the LLM's parameters. Here we will define a tool to search the Huggingface documentation for relevant information.

Replace the text "## YOUR TOOL DESCRIPTION HERE##" with a description of the contents of the database and when the LLM should make use of these tools. The database contains information about the documentation of the HuggingFace python module, scraped from their website.

[ ]:
```
from langchain.tools.retriever import create_retriever_tool

# Turn the vectorstore into a tool
retriever = vectorstore.as_retriever(search_kwargs = dict(k = 1))

retriever_tool = create_retriever_tool(retriever, "search docs", "## YOUR TOOL␣
 ↪DESCRIPTION HERE ##")
```

10 4. Connecting everything together

10.1 4.a. Set up the agent prompt (40 points)

Fill out the system prompt such that it would create an agent that answers questions pertaining to NLP and the Huggingface Transformers library. The prompt provided is a mostly standard ReAct prompt (LangChain slides 17-19), but is formatted to work with LangChain's JSON agent parsing. All you have to do is provide some additional context for how you expect the model to behave by replacing "## YOUR PROMPT HERE ##".

[ ]:
```
# Produce a system message for this component.

system_message = """[INST]<<SYS>>## YOUR PROMPT HERE ##
TOOLS:
------
Where appropriate, Assistant must use tools to access information that is␣
 ↪needed to complete User's request. The tools Assistant can use are shown␣
 ↪below along with their descriptions:
{tools}

FORMAT:
------
To use a tool, Assistant must use the following format:

Thought: Do I need to use a tool? Yes
```
{{

  "action": $TOOL_NAME, //should be one of [{tool_names}]

  "action_input": $INPUT //the input to the action

}}
```
Observation: the result of the action

When you are ready to respond to the user, use the following format:

Thought: Do I need to use a tool? No
```
  {{
    "action": "Final Answer",
    "action_input": $FINAL_ANSWER //the final answer to the user's request
  }}
```
<</SYS>>[/INST]
"""
```

```
instructions = """
[INST]
Conversation history:
{chat_history}
{input}
[/INST]
Thought: Do I need to use a tool?{agent_scratchpad}
"""
```

[ ]:
```
from langchain_core.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
    SystemMessagePromptTemplate,
)

from langchain_core.messages import AIMessage, HumanMessage

# Sets up your prompt with few-shot examples for proper formatting.
# If you are utilizing Mixtral, combine the system_message and the first␣
 ↪HumanMessage.
# Mixtral does not have a system message.
messages = [
      SystemMessagePromptTemplate.from_template(system_message),
      HumanMessage(content="Before we begin, let's go through some examples."),
      AIMessage(content="I am ready!"),
      HumanMessage(content="Hello, I would like to know more about the␣
 ↪Huggingface library.", example=True),
      AIMessage(
                content="Do I need to use a tool? Yes\n" \
                "\n```\n{\"action\": \"search docs\", \"action_input\":␣
 ↪\"Huggingface library\"}\n```" \
                "\nObservation: https://huggingface.co/docs/transformers/
 ↪index\nThe Huggingface library is a repository of pre-trained models for␣
 ↪Natural Language Processing (NLP) and contains a variety of language models,␣
 ↪tokenizers, and other tools for working with NLP and other machine learning␣
 ↪tasks."/doc> \
                "\n\nThought: Do I need to use a tool? No\n" \
                "\n```\n{\"action\": \"Final Answer\", \"action_input\": \"The␣
 ↪Huggingface library is a repository of pre-trained models for Natural␣
 ↪Language Processing (NLP) and contains a variety of language models,␣
 ↪tokenizers, and other tools for working with NLP and other machine learning␣
 ↪tasks. Source: https://huggingface.co/docs/transformers/index\"}\n```",
                example=True
                ),
       HumanMessage(content="Is it true that the Huggingface library hosts␣
 ↪audio and vision models too?", example=True),
       AIMessage(
```

```
                content="Do I need to use a tool? Yes\n" \
                "\n```\n{\"action\": \"lfact_check", \"action_input\": \"The␣
↪Huggingface library hosts audio and vision models too.\"}\n```" \
                "\nObservation: True. Explanation: The Huggingface library␣
↪hosts audio computer vision, and multimodal models in addition to language␣
↪models and tokenizers. This is true given the information from the following␣
↪source: https://huggingface.co/docs/transformers/index" \
                "\n\nThought: Do I need to use a tool? No\n" \
                "\n```\n{\"action\": \"Final Answer\", \"action_input\": \"True.
↪ The Huggingface library hosts audio computer vision, and multimodal models␣
↪in addition to language models and tokenizers. This is true given the␣
↪information from the following source: https://huggingface.co/docs/
↪transformers/index\"}\n```",
                example=True
                ),
        HumanMessage(content="Thank you!", example=True),
        AIMessage(content = "Do I need to use a tool? No\n```\n{\"action\":␣
↪\"Final Answer\", \"action_input\": \"You're welcome!\"}\n```",␣
↪example=True),
        HumanMessagePromptTemplate.from_template(template=instructions),
    ]
```

10.2 4.b. Set up the agent (20 points)

Utilize the `create_json_chat_agent(llm=llm, tools=tools, prompt=prompt)` function of the LangChain library to create an agent to utilize the given prompt. This creation should take place in the `agent_create(context, instructions, tools)` function which takes as it parameters a system messages `system`, instructions to the model (to be put in the user message) `instructions`, and a list of tools `tools`.

Pass your agent into the `AgentExecutor` constructor. Notice how limits have been placed to avoid infinite recursion with `max_iterations`, and the ability to remember past turns of the conversation through `ConversationBufferMemory` have been added.

Hint: prompt in create_json_chat_agent needs to be a ChatPromptTemplate.

[ ]:
```
from langchain.agents import AgentExecutor, create_json_chat_agent
from langchain.memory import ConversationBufferMemory

def agent_create(messages, tools):
    agent = None

    llm = ChatOpenAI(api_key = api_key, base_url = base_url,
                 model = model, temperature= 0, max_tokens = 1000)

    ## YOUR CODE HERE ##
    # Produce a ChatPromptTemplate with the instructions.
```

```
    # Create a JSON agent with the llm, tools, and prompt.


    # Creates an AgentExecutor wrapping around the agent, along with a␣
 ↪conversation memory.
    memory = ConversationBufferMemory(memory_key='chat_history', max_length=3)
    agent_executor = AgentExecutor(
                                agent=agent, tools=tools, verbose=True,
                                max_iterations=3, handle_parsing_errors=True,␣
 ↪memory=memory
                               )
    return agent_executor

tools = [retriever_tool, fact_check]


# If you want to restart the conversation from scratch, you must run this cell␣
 ↪again.
agent_executor = agent_create(messages, tools)
```

11 5.a. Run the agent (20 points)

Create an AgentExecutor object containing your agent and tools by defining a function `run_agent(agent, tools)`, then use the `invoke(input)` method to run it. In the markdown cell below **in bold**, record 3 conversation turns with the LangChain assistant. A conversation turn includes both your input and the output from the agent.

Note: You will be graded on the quality of your input and your conversation turns, not the output of the model. Ask the model interesting questions, see where it has to utilize the database, when it utilizes the fact checker, or when it is able to respond all on its own. You have free reign to test where your model succeeds and where it fails here.

An example of invoking an assistant can be seen in the LangChain examples provided on the course website.

[ ]:
```
def run_agent(agent, tools):
    inputDict = dict()

    userInput = input('User: ')

    inputDict = {'input': userInput}

    ## YOUR CODE HERE ##
    # Return the output of the agent executor for the given user input.
```

```
run_agent(agent_executor, tools)
```

Enter markdown here

12 [Bonus Points] Analysis (10 points)

In the markdown cell below, in bold, record 5 (additional) interesting conversation turns with the model, along with annotations as to why you find the conversation interesting. Try to get a mixture of examples, some negative and some positive.

Enter markdown here

13 5.b. [Bonus Points] Utilize the ArXiv API tool (15 points)

Add additional instructions to your previous instructions user message, instructing the agent to find relevant papers utilizing the ArXiv API tool to append to its response. If no papers can be found, have the model respond with "I could not find any relevant papers on the topic." If a paper can be found, have the model respond with the title and abstract of the paper. Execute your agent with a statement about the huggingface library with your assistant in which the assistant gives you a reference to a paper. In the markdown cell below, keep track of some interesting links to papers you found while utilizing your LangChain agent.

Hint: Examples of how to load tools are seen both in this document and in the LangChain examples available on the course website, specifically the example with the internet search agent

[ ]:
```
## YOUR CODE HERE ##
# Add the 'arxiv' tool to the list of tools with load_tools


# Create an agent using your previous functions, then run it.
```

Enter markdown here

14 Bonus Points

Any non-trivial task that is relevant for this assignment will be considered for bonus points. For example:

1. Make the LangChain assistant work well for different language models. Keep track of interesting differences between how the models respond. Do they have a consistent authorial voice? What are some ways you think that you could tell the models apart from one another?
   - What insights do you think this could give you into how these models were trained?
2. Find another tool that you feel would be interesting to use with this NLP assistant. Add this tool to your LangChain assistant, and show some examples of your conversations with the tool.
   - Make sure to explain why you feel this tool would further improve a system designed to give information about NLP and the Huggingface library

3. Some of you may have noticed that on occasion that LanceDB will respond with non-English pages from the HuggingFace documentation. Try to use an LLM as a tool for translation, and connect it with your overall system.
    - It is better to make a tool than to try and have the overall model try to perform the translation.
    - If you can get the model to perform it without an additional tool, that is still a valid solution. However, you would have to defend against the model getting confused and starting to respond in the wrong language.
4. Try rewriting the agent you created in this assignment in Microsoft's AutoGen framework. You do not have to recreate any extra credit that you completed.