ITCS 4111/5111: Introduction to NLP

LangChain for Building LLM-powered Applications

Justin Smith, with help from Erfan and Razvan

jsmit840@charlotte.edu

Department of Computer Science @ CCI

Learning Objectives

- What is the LangChain Library?
- What are LangChain components?
  - How do you integrate them with each other?
- How does a LangChain agent work?
- For what cases does LangChain work?
- Under what circumstances does LangChain fail?

What is LangChain?

- A library that is designed to ease the development of applications that utilize both an **LLM** and **external tools**.
  - More powerful applications than a single prompt: **dialogue systems** (next homework), **data science** (query a database), **multimodal input** (send preprocessed audio/image/video to the LLM).

- **Central idea:** "Chain" together the LLM and, optionally, additional components.
  - Similar to a linked list, the "chain" is sequential.
  - Dissimilar to a linked list, the input to the next component is cumulative

LangChain Core Concepts

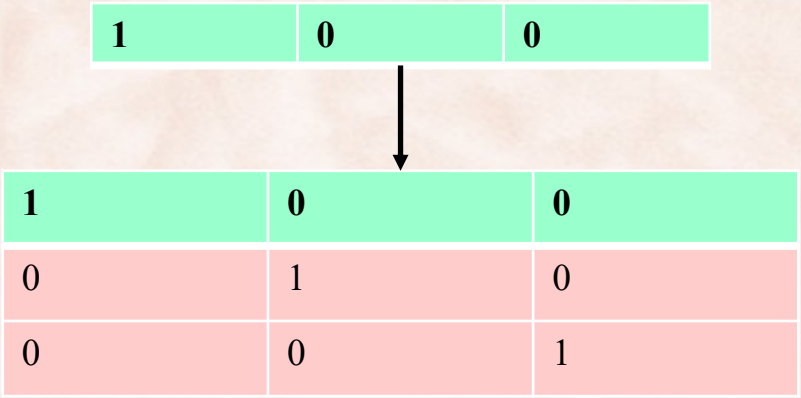
- **LLMs**: Large language models like GPT-3.5, GPT-4, Llama2, Mixtral.
- **Prompt templates**: A template for building a chain of prompts
  - The output of one prompt can be input into a different prompt.
- **Memory**: Keeps track of past messages and information from external **tools**.
- **Agents**: Abstraction of the chain of LLM prompts and components.
  - To the user, a chain of prompts is single system to interact with.
- **Vector Stores:** Databases that store the meaning of various text documents.
- **Tools**.

Common LangChain **Components**

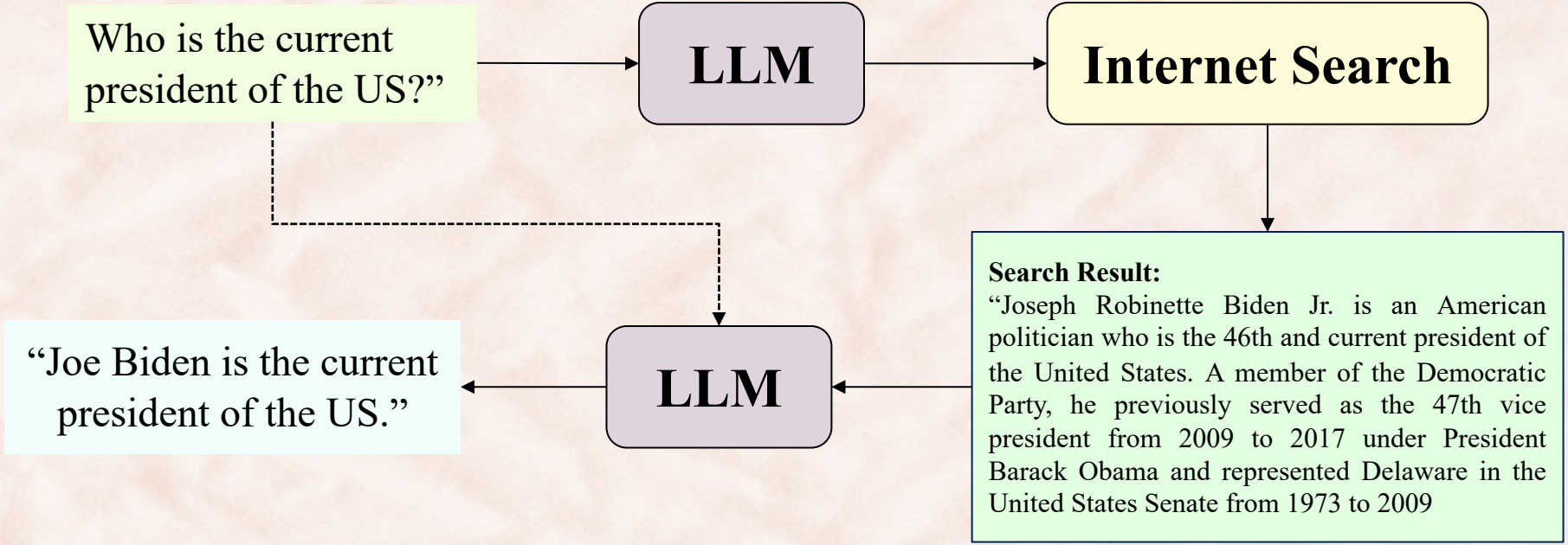
https://python.langchain.com/docs/integrations/components

- **LLMs**.
- **Vector Stores**:
  - A type of database that stores data as high-dimensional vectors
  - This data can be text, images, video, and more.
  - Why do we use vector stores? Find most similar:
    - *documents* for a query.
    - training *examples* for a test example.
- **Tools**:
  - Web Search (this lecture).
  - Calculators, Compilers, Interpreters, …
  - Image Generation, File Access, …

Example: Vector store with *one-hot encodings*



The Chain in Lang**Chain**



The Chain in Lang**Chain**

- **PromptTemplate** class:
  - Sets up a prompt with input variables, like a formatted string.
  - Make an instance of the prompt with the method format(inputs).
- **LLM** class:
  - Contains a method to call an LLM.
- **LLMChain** class:
  - Takes in an LLM as an object, along with a prompt template.
  - Has a method, invoke(inputs), that will get the LLM response.
- **Agent** class:
  - Abstract an LLM, along with other components and a prompt, into a single object.
- **AgentExecutor** class:
  - Method, invoke(inputs), will get the response of the entire chain.

LangChain for ReAct Prompting

- ReAct prompting is an approach for enhancing the decision making of LLMs, for example, choosing which tools to use for a given situation.
- The structure of the ReAct Loop:
  - <question> (<thought> <action> <action input> <observation>)* <thought> <answer>

Scenario 1

Question: Who is the current president of the US?

Thought: I should use a search engine to find relevant data.
Action: Open Google
Action Input: "Who is the current president of the US?"
Observation: The current president is Joe Biden.

Thought: I now know the final answer.
Final Answer: Joe Biden

Scenario 2

Question: What is 2 + 2?

Thought: I now know the final answer.
Final Answer: 4

ReAct Prompting

- A style of prompting that can be used to let an LLM interact with external tools.
- This utilizes a format that is simple to parse:
  - **Question**: The input to the model.
  - **Thought**: A place for the model to state intentions, which tends to increase performance.
  - **Action**: The tool (if any) does the model wants to use.
    - **Action Input**: The input to the tool.
  - **Observation**: The output of the tool.
- The final answer is given in the format:
  - **Thought**: "I believe I have the final answer."
  - **Final Answer**: The answer the model gives to the user.

The Chain in Lang**Chain**

- **How do these components work together?**
  - input_variable is a parameter of the PromptTemplate class
  - This allows linking the output to one component with the input of another.

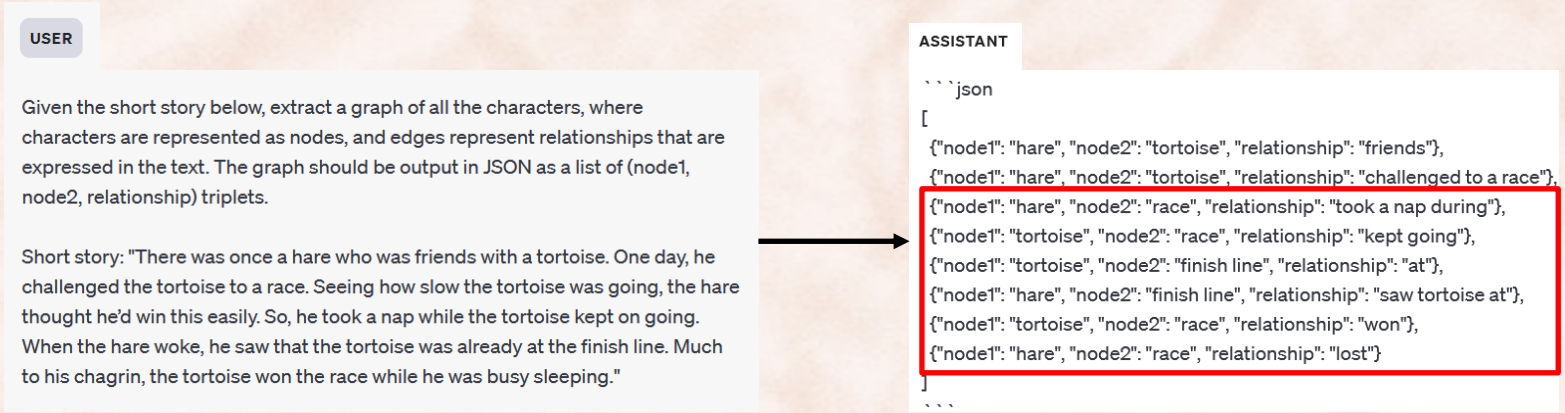
- **What format style should we use for exporting information?**
  - JSON is a great format to output in.
  - Many LMs can comprehend and produce JSON consistently.
  - Sometimes, just a string is sufficient.

LangChain Example

- Jupyter notebook.

Zero-Shot

- Zero-shot refers to giving the LLM no examples, only the instructions:
  - This is sufficient for some tasks, but sometimes it is hard to clearly express the intended output in English, e.g. output format.



OpenAI Playground example here: [Zero-Shot Example with GPT-4-Turbo]

In-context Learning: Few-Shot Examples

Language Models are Few-Shot Learners

- Sometimes it's better to give a few examples to the LLM to explain the problem.
  - For example, 1-shot refers to one example, 2-shot to two examples, and so on.
  - In-context examples tend to increase the efficiency of the model.
- In general, this is referred to as *few-shot*.

- Let's try the same prompt as before, but with examples of what we want the model to produce.

OpenAI Playground example here: Few-shot example with GPT-4-Turbo

Examples and Improved Output

USER

Given a short story, extract a graph of all the characters, where characters are represented as nodes, and edges represent relationships that are expressed in the text. The graph should be output in JSON as a list of (node1, node2, relationship) triplets.

For example, if the short story is:
"Red Riding Hood had lunch at her grandmother's place. Then she parted ways with her grandmother and walked into the woods. She met and talked with the Wolf, who followed her deep into the forest.",
then the output should be:

```
[
  ["Red Riding Hood", "grandmother", "parted ways with"],
  ["Red Riding Hood", "Wolf", "met"],
  ["Red Riding Hood", "Wolf", "talked"],
  ["Wolf", "Red Riding Hood", "followed"]
]
```

If the short story is:
"The raccoon brushed his teeth at 5pm, while the elephant got ready at 6pm. The raccoon met with the elephant at 7pm for dinner. The elephant got a steak, the raccoon got mashed potatoes. The elephant went home at 9pm, the raccoon went home at 10pm. Later, the elephant made a phone call to the raccoon.",
then the output should be:

```
[
  ["raccoon", "elephant", "met at 7pm for dinner"]
  ["elephant", "raccoon", "made a phone call"]
]
```

Now output the graph for this short story: "There was once a hare who was friends with a tortoise. One day, he challenged the tortoise to a race. Seeing how slow the tortoise was going, the hare thought he'd win this easily. So, he took a nap while the tortoise kept on going. When the hare woke, he saw that the tortoise was already at the finish line. Much to his chagrin, the tortoise won the race while he was busy sleeping."

ASSISTANT

```
[
  ["hare", "tortoise", "friends with"],
  ["hare", "tortoise", "challenged to a race"],
  ["hare", "tortoise", "took a nap while"],
  ["tortoise", "hare", "kept on going"],
  ["hare", "tortoise", "saw at the finish line"],
  ["tortoise", "hare", "won the race"]
]
```

Critique of LangChain

- LangChain is unnecessary when using solely the LLM.
  - Instead, easier to just keep track of the conversation in the code.
- Tools are not always the simplest, most effective solution.
  - It might be enough to provide examples to the LLM and have it accomplish the task.
- LangChain does not improve the base model.
- LangChain cannot improve the context length.
  - No language model can keep an infinitely long conversation going.
  - Sometimes a longer prompt outperforms a long list of function calls.

Learning Objectives

- What is the LangChain Library?
- What are LangChain components?
  - How do you integrate them with each other?
- How does a LangChain agent work?
- For what cases does LangChain work?
- Under what circumstances does LangChain fail?

Frameworks for deploying LLM-based Web Apps

- Streamlit:
  - **Pros**: Python-based, more configurable than Chainlit (e.g. authentication), more UI components than Chainlit and Gradio, and easy to deploy for free on a public URL through share.streamlit.io (through a .edu account)
  - **Cons**: Needs additional work to support token streaming and does not support visualizing LangChain prompt chains.
- Chainlit:
  - **Pros**: Python-based, easily visualizes prompt chains from LangChain (as of version 0.2.0), and supports token streaming.
  - **Cons**: Needs some work to deploy and does not offer free hosting that can be accessed through a public URL. To host on a public URL students would need to use a cloud provider like Heroku.

Frameworks for deploying LLM-based Web Apps

- Chat UI:
  - **Pros**: Easily integratable with a a multitude of LLM hosting services and frameworks such as Azure, HuggingFace Inference endpoints, and vLLM. Full UI and deployment configurability. Chat-UI is a replica of HuggingChat.
  - **Cons**: Needs JavaScript to customize, does not visualize LangChain prompt chains.

- Gradio:
  - **Pros**: Python-based, more UI components than Chainlit, and easy to deploy to a public URL through Huggingface spaces.
  - **Cons**: Does not support visualizing LangChain prompt chains.

Recommended Readings

- https://learn.deeplearning.ai/langchain
- https://python.langchain.com/docs/get_started/introduction