CS 5316: NLP Theory and Applications

PA4 Manual

1 What is LangChain?

A powerful framework for building scalable and robust LLM applications

At first glance, building an LLM application might seem straightforward: all you need is an LLM and a prompt. However, modern LLM-based applications are far more sophisticated. They often need to:

- Access and integrate external tools or APIs
- Retrieve and utilize memory from past interactions
- Make complex reasoning or multi-step decisions

LangChain provides a simple yet highly flexible framework that addresses these requirements. Now widely used in both industrial and research settings, LangChain allows developers to design, orchestrate, and scale LLM applications efficiently, turning basic prompt-response models into scalable applications.

2 Assignment Structure

The assignment is structured as follows:

- **Part 1: Building LLM pipelines**
  This section would equip students with a deep understanding of using LCEL (LangChain Expression Language) and applying it to different tasks including prompt engineering, RAG, AI-as-a-Judge, etc.
- **Part 2: Agentic Workflows**
  This section would employ the use of function/tool calling to create agents that perform a set of tasks.
- **Part 3: Building a Chatbot**
  This section would focus on building a complete sophisticated chatbot with an interface.

The source code for **Parts 1 and 2** is provided to you. However, students are required to independently develop and submit the code for **Part 3**. All corresponding files for **Part 3** must be placed within the `/Part3` directory. The detailed problem statement for Task 3 can be found in Section 6.

This assignment will utilize the **Google Gemini 2.0-Flash** model (unless otherwise specified). To access this model, you will need to obtain an **API key** from the official Google website. Ensure that your key is configured correctly before running any code.

2.1 Guidelines

- **LCEL Syntax:** All tasks must be implemented using the **LangChain Expression Language (LCEL)** syntax.
- **Library Consistency:** Use the same libraries and functions demonstrated in the manual for Parts 1 and 2. While alternative implementations exist, using them may indicate insufficient understanding of the core material.
- **Context Engineering:** After Part 1, Task 1, all chains must adhere to the principles of **Context Engineering** with **System Messages**, as described in the manual. Non-compliance will result in a deduction of marks.
- **Reflective Questions:** All reflective questions must be completed independently. The use of AI tools or external assistance is strictly prohibited for solving them.

Note: It is **highly recommended** for the students to go through the manual, and to solve the given code examples on their own before attending the assignment.

3 Building LLM pipelines

3.1 Prompt Templates

Now in order to initialize the prompt variable, we'll be using the **ChatPromptTemplate** library. A prompt template allows general structure for your prompt and leave placeholders, aka variables, that can be dynamically filled in later with different inputs.

For example, if we define the prompt template as "Tell me a fun fact about animal.", the placeholder animal acts as a variable that we can replace with any value later, like "dolphins", "cats", or "penguins" without rewriting the entire prompt.

This is very useful for tasks where an LLM's input depends on the output from another LLM. The output from one model can simply be inserted into the placeholder of another.

3.2 Chaining

One of the most powerful features of LangChain is the ability to connect multiple components together seamlessly. You can think of each step; a prompt, an LLM, a parser, a retriever, as a small, independent component. Each one does a specific job. But the real magic happens when you chain them together.

Chaining allows the output of one component to seamlessly flow into the next. For example, a PromptTemplate might format your question, then pass it to an LLM for generation, and finally a Parser cleans up the response.

This is done via a piping (|) operator. The pipe connects multiple components into a single, logical pipeline (just like Unix pipes). A simple chain, for example, might be:

```

chain = prompt | llm | parser

```

Where we define the prompt (using a prompt template), pass it to an LLM, and then to a parser. The chain is then invoked to get the final response. This chaining operation is an integral component for the LCEL (LangChain Expression Language).

A sample code for invoking a chain is given below:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import
↪  StrOutputParser
from langchain_google_genai import
↪  ChatGoogleGenerativeAI

prompt = ChatPromptTemplate.from_template("Tell me a
↪  fun fact about {animal}.")

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash",
↪  temperature=0)
parser = StrOutputParser()
chain = prompt | llm | parser

```

```python
print(chain.invoke({"animal": "dolphins"}))

# 1. The Prompt takes your variable and formats the
↪  input: "Tell me a fun fact about dolphins."
# 2. The LLM (Gemini) receives that prompt and
↪  generates an answer.
# 3. The Parser cleans the output so you just get the
↪  text response.
```

3.3 Context Engineering with System Messages

When working with LLMs in LangChain, you often see two types of messages: **System Messages** and **User/Human Messages**.

For a structured system, we'll define the System messages to be the **ground rules** and the **role** of the LLM, providing context for the task at hand, while user messages are usually the **actual inputs** passed to the LLM.

Providing context almost always improves the quality of the LLM's responses. For example, if our task is to translate text, it's better to first frame the model as a translator rather than jumping straight into the request. This step of providing context via System Messages is a subset of the vast field of Context Engineering.

For example, for a language translation task, we'll define these messages as follows:

System Message (role / context):

”You are a professional multilingual translation expert with native-level proficiency in 50+ languages. Your role is to provide accurate, natural-sounding translations while preserving the original meaning, tone, and cultural context.

Translation Guidelines:
- Maintain the original intent and nuance of the source text
- Preserve formal/informal tone matching the source
- Ensure grammatical correctness and natural flow in the target language
- Flag any ambiguous phrases or potential translation challenges”

Sample code for this is provided below:

```python
# System message: setting the role and rules. The
↪  message has been trimmed for brevity.
system_msg = """
```

```
You are a professional multilingual translation
 ↪  expert...
"""

user_msg = """
Please translate the following business email from
 ↪  English to Spanish:
{email}
"""


prompt = ChatPromptTemplate.from_messages([
    ("system", system_msg),
    ("user", user_msg)
])

# Assume that you have an email below
email = ...

chain = prompt | llm
print(chain.invoke({"email": email}).content)

```

Note that we used the from_template() function before. However, if you're using System Messages, then you'll use the from_messages() function.

Note that we can also link multiple chains together via the piping operator to execute them in a sequence.

3.4 Parallelization — Running Multiple Chains Together

Now that we've established how to invoke chains, we can use them to divide a larger problem into shorter, sub-problems.

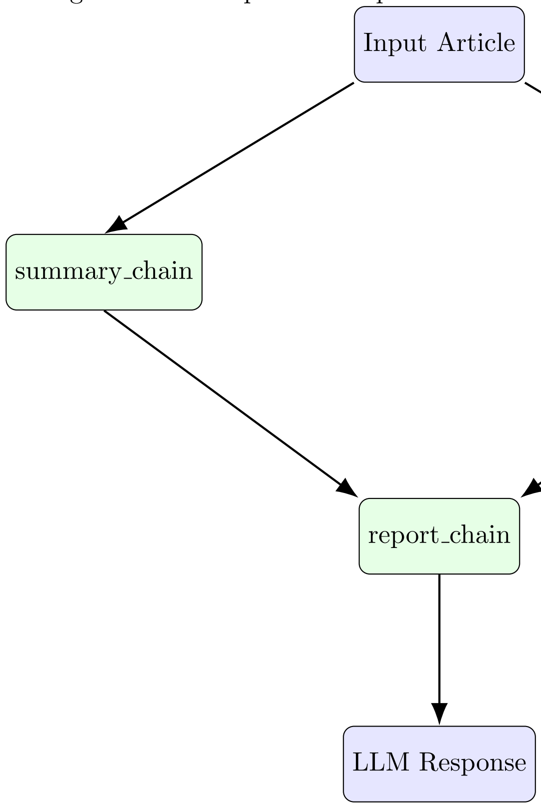
So for example, if you wanted to create an Article Analysis System (a system that generates a report summarizing an article's key points and overall sentiment), you can divide it into the following chains:

- Summary Chain: Summarizes the input article in 2-3 sentences, extracting the key points for easy consumption.
- Sentiment Chain: Analyzes the sentiment of the article and classifies it as Positive, Neutral, or Negative.

- Report Generation Chain: Takes the outputs of both the Summary Chain and the Sentiment Chain as inputs, and produces a concise report that combines the key points and the sentiment insights.

In this setup, the Summary Chain and the Sentiment Chain can run in parallel, and their outputs are then "fanned in" to the Report Generation Chain, which integrates the results to produce the final output.

A diagram for this operation is provided below:



The code for this is as follows. We'll first start by defining our chains:

```
summary_prompt = ChatPromptTemplate.from_template(
    "Summarize the following article in 2-3
    ↪  sentences:\n\n{article}"
)
summary_chain = summary_prompt | llm |
↪  StrOutputParser()

sentiment_prompt = ChatPromptTemplate.from_template(
```

```python
    "Analyze the sentiment of the following text and output Positive, Neutral, or Negative:\n\n{article}"
)
sentiment_chain = sentiment_prompt | llm | StrOutputParser()

report_prompt = ChatPromptTemplate.from_template(
    "Using the summary:\n{summary_chain_output}\nAnd the sentiment: {sentiment_chain_output}\nWrite a concise report for a manager."
)
report_chain = report_prompt | llm | StrOutputParser()
```

Now that we have initialized our chains, we'll need to link them and run them in parallel. This is achieved through LangChain's RunnableMap as follows:

```python

first_map = RunnableMap({
    "summary_chain_output": summary_chain,
    "sentiment_chain_output": sentiment_chain
})

```

Invoking this map would allow us to run both chains in parallel, and each chain's output will be saved under its corresponding key.

Next, we want to use the outputs from both chains as inputs to a final report_chain. Since the outputs for the summary_chain and sentiment_chain are already computed, we use RunnablePassthrough() to pass them along unchanged. We now get our final pipeline as follows:

```python
second_map = RunnableMap({
    "summary_chain_output": RunnablePassthrough(),
    "sentiment_chain_output": RunnablePassthrough(),
    "report_chain_output": report_chain
})

pipeline = first_map | second_map
```

4 RAG (Retrieval Augmented Generation)

Have you ever asked ChatGPT a question and thought, “Hmm… that sounds confident, but is it actually true?” That’s one of the biggest challenges with language models, they’re great at sounding right, even when they’re not. What if our model could look things up before answering?

RAG is a technique that combines two powerful components:

- Retrieval: Fetching relevant, factual information from a knowledge source (e.g., documents, PDFs, databases, or websites).
- Generation: Using an LLM to create context-driven responses based on both the retrieved context and the user query.

We normally use RAG when we need up-to-date answers to a query (grounding the LLM) e.g. ”What’s the weather today?”, or when we need the LLM to answer with respect to a certain domain e.g. ”Based on Policy Documents 8, 9, 10 and 15, how many annual leaves are allowed for HR staff?”. A regular LLM wouldn’t know what those policy documents are, and it’s not practical (or cheap) to attach all of them to every query. A better approach is to store all your documents in a database, and when a question comes in, retrieve only the most relevant pieces of information to include in the model’s prompt.

While fine-tuning a model on your documents could achieve something similar, it’s expensive and time-consuming. RAG provides a much easier alternative: we simply prepend the relevant context to the model’s prompt before generation.

But this raises an important question: how do we decide what’s relevant?

That’s where vector databases (or vector stores) come in. They store vector embeddings, numerical representations of the meaning of each document chunk. When a user asks a question, that query is also converted into a vector, and the database finds the most semantically similar chunks to include in the model’s context.

5 Function/Tool Calling for Agents

This is a mechanism that provides an LLM with ”tools” (functions) to perform a variety of tasks. This could range from performing web search, calling APIs, or accessing an external calculator, all from within a conversational context.

When using tool calling, you define Python functions as `@tool`-decorated components. Each tool has 3 essential components:

- A **name** (the function name)
- A **description** (taken from the docstring)
- A set of **input parameters** (with type hints)

The LLM, when connected through LangChain's agent architecture, can read the tool definitions, decide which one to invoke, and automatically call them to complete the user's query.

As an example, if you want the LLM to calculate some marks, you can create a function/tool for it as follows:

```python

@tool
def calculate_percentage_marks(marks_obtained: int, marks_total: int) -> float:
    """Calculate the percentage using marks obtained and total marks."""
    return round((marks_obtained / marks_total) * 100, 2)

```

Note how we used the three essential components in this example.

Now in order for our LLM to access these tools, we do as follows:

```python

@tool
def calculate_percentage_marks(marks_obtained: int, marks_total: int) -> float:
    """Calculate the percentage using marks obtained and total marks."""
    return round((marks_obtained / marks_total) * 100, 2)

@tool
def check_current_time() -> str:
    """Check the current system time."""
    import time
    return time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())


tools = [calculate_percentage_marks, check_current_time]

```

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a household assistant that can
     ↪  use the provided tools to answer questions and
     ↪  perform actions."),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}")
])

agent = create_tool_calling_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent,
 ↪  tools=tools, verbose=True)

queries = [
    "Calculate the percentage of 650 marks out of
     ↪  800.",
    "What time is it right now?"
]

for q in queries:
    print(f"\n--- Query: {q} ---")
    result = agent_executor.invoke({"input": q})
    print("Final Output:", result["output"])
```

6 Building Chatbots

In this section, your task is to build a chatbot using **LangChain** with a **Streamlit** interface. The chatbot should implement **RAG-Fusion**, a modified version of Retrieval-Augmented Generation (RAG). The paper for this concept is linked ahead: https://arxiv.org/pdf/2402.03367

6.1 Routing Queries to Appropriate Chains

Once a user query is refined and passed through the RAG-Fusion module, the LLM must select the appropriate chain based on the task. The following chains should be implemented:

- **Summarization Chain:** For queries asking for summaries of sections or topics. This chain retrieves relevant chunks and summarizes them into concise, readable answers.
- **Code Generation Chain:** For queries related to generating source code.

- **Web Search Chain:** For queries that require searching information on the web.

6.2 Dynamic Chain Creation

If the LLM determines that an additional chain is necessary:

1. An **Agent** should generate the source code for the new tool.
2. A separate **Agent** should evaluate the tool for correctness and reliability.
3. Once validated, add the new chain to the list of available chains for the LLM to route queries to.

6.3 LLM Critic Evaluation

Implement an **LLM Critic** to evaluate outputs from the following systems and assign a performance score:

- RAG
- RAG-Fusion
- RAG-Fusion with dynamic routing of chains

Comment on the results and insights gained from the comparisons. Which of the three has the highest reward, and why? Also mention if Reciprocal Rank Fusion was proved to be useful in this scenario.

6.4 Requirements

Your Streamlit interface should include:

- A button to switch between different forms of RAG.
- A button to upload PDF documents.
- Streaming messages to display conversation in real-time.
- Storage of conversation history.

For the chains:

- They should be well defined, adhering to the principle of Context Engineering
- Assigning the correct chain to each task is critical. Failure to execute a task using the appropriate chain will result in a loss of marks.

The output from the LLM Critic should be displayed in the **terminal** rather than within the Streamlit interface.