LangChain Framework

NLP Course Presentation

Name: Alireza Dastmalchi Saei

Stu No.: 993613026

Contents

1 Introduction

LangChain is a framework designed for developing applications powered by large language models (LLMs). It streamlines every phase of the LLM application lifecycle, including development, productionization, and deployment.

The aim of this presentation is to introduce class members to the LangChain framework and illustrate how to implement a chatbot using its features. Topics covered in this presentation include various types of memories in chatbots, Chains, Evaluation methods, and Agents.

In the second part, a step-by-step process of implementing a chatbot is provided, which includes Document Loading, Document Splitting, Vectorstores and Embeddings, Retrieval, QnA of custom document, and Chatting with Document.

2 LangChain Framework

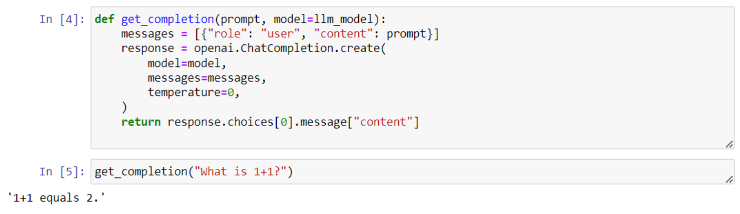
LangChain is a framework tailored for applications leveraging large language models (LLMs). Its primary goal is to provide a development environment for LLM applications, addressing key stages from development to deployment.

2.1 Models, Prompts, and Output Parsers

LangChain supports both direct API calls to OpenAI and enhanced functionality via its integrated prompts, models, and output parsers. This integration streamlines the development process, allowing for more sophisticated and customizable interactions with LLMs.

2.1.1 Direct API Calls

Direct API calls enable straightforward interactions with OpenAI's models. This approach is quick and effective for simple use cases but lacks the flexibility and advanced features provided by LangChain. (For invoking a non-local model, an OpenAI API Token is required!)



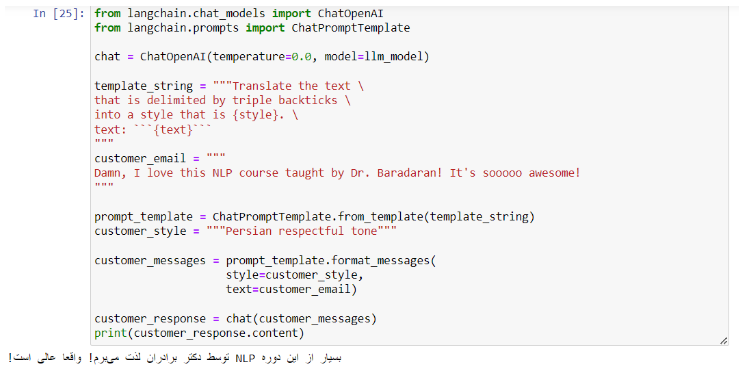
```
In [4]: def get_completion(prompt, model=llm_model):
            messages = [{"role": "user", "content": prompt}]
            response = openai.ChatCompletion.create(
                model=model,
                messages=messages,
                temperature=0,
            )
            return response.choices[0].message["content"]

In [5]: get_completion("What is 1+1?")

'1+1 equals 2.'
```

Figure 1: Direct API Calls

2.1.2 API Calls through LangChain

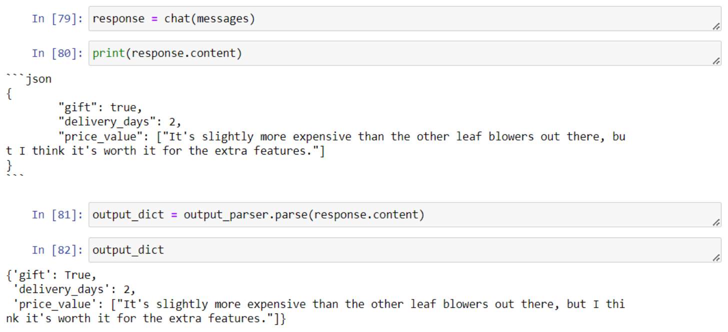
LangChain enhances API calls by incorporating prompts, models, and output parsers:

- **Prompts:** Customizable prompts that guide the LLM in generating desired responses.
- **Models:** Predefined and custom models that can be integrated together.
- **Output Parsers:** Tools to process and structure the raw output from LLMs, making it more usable for specific applications.



```
In [25]: from langchain.chat_models import ChatOpenAI
         from langchain.prompts import ChatPromptTemplate

         chat = ChatOpenAI(temperature=0.0, model=llm_model)

         template_string = """Translate the text \
         that is delimited by triple backticks \
         into a style that is {style}. \
         text: ```{text}```
         """
         customer_email = """
         Damn, I love this NLP course taught by Dr. Baradaran! It's sooooo awesome!
         """

         prompt_template = ChatPromptTemplate.from_template(template_string)
         customer_style = """Persian respectful tone"""

         customer_messages = prompt_template.format_messages(
                             style=customer_style,
                             text=customer_email)

         customer_response = chat(customer_messages)
         print(customer_response.content)
بسیار از این دوره NLP توسط دکتر برادران لذت می‌برم! واقعا عالی است!
```

Figure 2: Direct API Calls



~~~
In [79]: response = chat(messages)

In [80]: print(response.content)

```json
{
        "gift": true,
        "delivery_days": 2,
        "price_value": ["It's slightly more expensive than the other leaf blowers out there, bu
t I think it's worth it for the extra features."]
}
```

In [81]: output_dict = output_parser.parse(response.content)

In [82]: output_dict

{'gift': True,
 'delivery_days': 2,
 'price_value': ["It's slightly more expensive than the other leaf blowers out there, but I thi
nk it's worth it for the extra features."]}
~~~

Figure 3: Parser Usage

2.2 Memory

LLMs are inherently stateless, meaning each transaction is independent of the previous ones. However, for chatbots to simulate memory, the entire conversation context must be provided with each interaction. LangChain offers various memory types to manage and accumulate this conversation context effectively:

1. **ConversationBufferMemory**

   Stores the entire conversation history in a buffer, which is included in every interaction. This method is simple but can become unwieldy with long conversations.

2. **ConversationBufferWindowMemory (k)**

   Maintains a sliding window of the last k interactions. This approach balances memory usage and context relevance by limiting the amount of retained conversation history.

3. **ConversationTokenBufferMemory**

   Accumulates conversation context up to a specified token limit. This method ensures the context remains within the LLM's token limit, optimizing memory usage without losing essential information.

4. **ConversationSummaryMemory**

   Summarizes the conversation history into a concise format, retaining the most relevant information. This method is efficient for long conversations, as it reduces the amount of context needed while maintaining coherence.

2.3 Chains

Chains in LangChain allow for the sequential processing of inputs through multiple steps, enabling complex workflows and decision-making processes:

1. **LLMChain**

   A single chain that processes input through a language model. It is straightforward and useful for simple tasks.

2. **Sequential Chains**

   These chains enable multi-step processing, where each step's output serves as the next step's input:

   – **SimpleSequentialChain:** A basic sequential chain with a fixed order of steps.

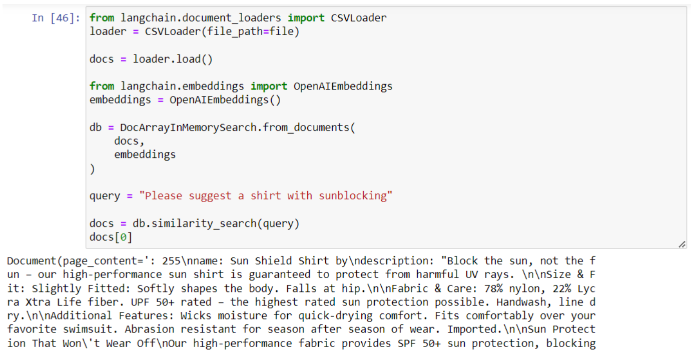
   – **SequentialChain:** A more flexible chain allowing for conditional branching and complex workflows.

3. **Router Chain**

   Directs inputs to different chains based on specific criteria. This chain type is useful for applications requiring dynamic decision-making.

2.4 Q&A over Documents

LangChain processes inputs through embedding techniques, transforming raw text into vector representations for efficient similarity searches across document repositories. By reading and interpreting data directly from documents, LangChain enhances the bot's ability to provide accurate and contextually relevant responses.



```
In [46]: from langchain.document_loaders import CSVLoader
         loader = CSVLoader(file_path=file)

         docs = loader.load()

         from langchain.embeddings import OpenAIEmbeddings
         embeddings = OpenAIEmbeddings()

         db = DocArrayInMemorySearch.from_documents(
             docs,
             embeddings
         )

         query = "Please suggest a shirt with sunblocking"

         docs = db.similarity_search(query)
         docs[0]
```

```
Document(page_content=': 255\nname: Sun Shield Shirt by\ndescription: "Block the sun, not the f
un – our high-performance sun shirt is guaranteed to protect from harmful UV rays. \n\nSize & F
it: Slightly Fitted: Softly shapes the body. Falls at hip.\n\nFabric & Care: 78% nylon, 22% Lyc
ra Xtra Life fiber. UPF 50+ rated – the highest rated sun protection possible. Handwash, line d
ry.\n\nAdditional Features: Wicks moisture for quick-drying comfort. Fits comfortably over your
favorite swimsuit. Abrasion resistant for season after season of wear. Imported.\n\nSun Protect
ion That Won\'t Wear Off\nOur high-performance fabric provides SPF 50+ sun protection, blocking
```

Figure 4: Simple QA System

2.5 Evaluation

LangChain provides several methods to evaluate and improve models, ensuring robust and accurate performance:

1. **Example Generation**

   Generates examples to test and refine models. These examples can be hard-coded or generated by the LLM.

2. **Manual Evaluation and Debugging**

   Involves human evaluators assessing the model's performance and identifying areas for improvement.

3. **LLM-assisted Evaluation**

   Utilizes LLMs to assist in the evaluation process, automating parts of the assessment and providing insights.

4. **LangChain Evaluation Platform**

   A dedicated platform for evaluating models, offering tools and metrics to measure performance comprehensively.

2.6 Agents

Agents in LangChain can utilize built-in tools or custom-defined tools to enhance functionality:

2.6.1 Built-in Tools

LangChain offers several pre-built tools for common tasks:

- **DuckDuckGo Search:** Enables web search capabilities.
- **Wikipedia:** Accesses information from Wikipedia.
- **Python (PythonREPLTool):** Executes Python code.

2.6.2 Custom Tools

Users can define their own tools to meet specific requirements, extending the framework's versatility.

3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) combines traditional information retrieval systems with generative LLMs, enhancing their ability to provide contextually relevant responses.

3.1 Overview

RAG frameworks leverage external datasets to retrieve contextual documents as part of the LLM's execution process. This approach is beneficial for applications requiring specific document references, such as technical support and content summarization.

3.2 Document Loading

Documents can be loaded from various sources, ensuring flexibility and broad applicability:

1. **PDF**

   Each page in a PDF is treated as a separate document, capturing its content and metadata.

2. **URLs**

   Documents can be retrieved from web URLs, enabling dynamic and up-to-date information access.

3. **YouTube**

   Video content can be transcribed and treated as documents, allowing for multimedia interaction.

4. **Notion**

   Integrates with Notion to retrieve documents, leveraging its robust content management capabilities.

5. **XML**

6. **JSON**

3.3 Document Splitting

Documents are split into smaller chunks while retaining meaningful relationships. This step is crucial for efficient processing and accurate retrieval. Various text splitters are available in LangChain to handle different types of documents:

1. **CharacterTextSplitter**

   Splits text based on character count.

2. **MarkdownHeaderTextSplitter**

   Splits text based on markdown headers.

3. **Context Aware Splitter**

   Maintains context by splitting at logical boundaries.

4. **TokenTextSplitter**

   Splits text into tokens, useful for LLMs with token-based context windows.

5. **SentencesTransformersTokenTextSplitter**

   Splits text using sentence transformers, enhancing context retention.

6. **RecursiveCharacterTextSplitter**

   Recursively splits text to maintain context.

7. **NLTKTextSplitter**

   Utilizes the Natural Language Toolkit (NLTK) for splitting.

8. **SpacyTextSplitter**

   Uses the Spacy library for text splitting.

3.4 Storage

Embeddings capture the context and meaning of text, enabling efficient and relevant information retrieval:

1. Embeddings

Embedding vectors represent the semantic meaning of text. Similar contexts will have similar vectors, facilitating accurate retrieval.

2. Vector Store

A storage system where all document splits are converted into embedding vectors. The vector store can find and return the top n similar embeddings to an input query. Duplicate data may cause the vector store to return duplicate answers, highlighting the need for careful data management.

3.5 Retrieval

Retrieval is the core of the RAG flow, enabling the system to find relevant information based on input queries:

1. Addressing Diversity

Maximum Marginal Relevance (MMR) fetches diverse results, enhancing the breadth of retrieved information.

2. Addressing Specificity

Metadata filters and self-query retrievers refine results based on specific criteria, ensuring relevance and accuracy.

3. Additional Tricks

Contextual compression retrievers compress results to fit within context limits, maintaining information integrity.

3.6 Chat with Data

LangChain supports interactive querying and chatting with documents through the RetrievalQA chain. This process involves using PromptTemplates and methods to resolve short context issues, such as Map Reduce, Refine, and Map Rerank, to ensure comprehensive and coherent responses.

4 Implementation

In this part, we will go through the code for implementing a chatbot to chat with custom uploaded document. The overview of the process can be seen below:

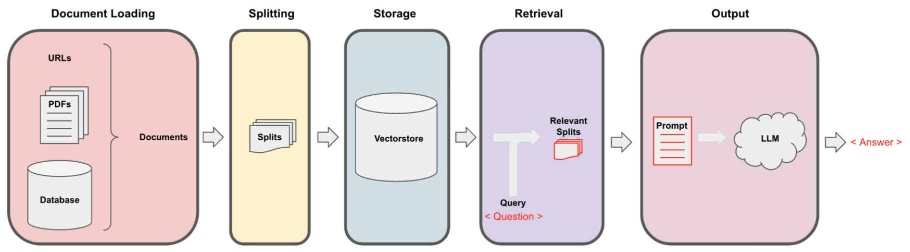


Figure 5: Implementation Process

4.1 Import required libraries

:

```python
import os
import openai
import sys
sys.path.append('../..')

import panel as pn  # GUI
pn.extension()

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

openai.api_key = os.environ['OPENAI_API_KEY']
```

Listing 1: Python Code for OpenAI API Key Setup

4.2 Custom Model as Conversational Agent

```python
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings

persist_directory = 'docs/chroma/'
embedding = OpenAIEmbeddings()
vectordb = Chroma(persist_directory=persist_directory,
    embedding_function=embedding)

question = "What are major topics for this class?"
docs = vectordb.similarity_search(question, k=3)
len(docs)

from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name=llm_name, temperature=0)
llm.predict("Hello world!")


from langchain.prompts import PromptTemplate
template = """Use the following pieces of context to answer
    the question at the end.
If you don't know the answer, just say that you don't know,
    don't try to make up an answer.
Use three sentences maximum. Keep the answer as concise as
    possible.
Always say "thanks for asking!" at the end of the answer.
{context}
Question: {question}
Helpful Answer:"""
QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context",
     "question"], template=template)

from langchain.chains import RetrievalQA
question = "Is probability a class topic?"
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.
                                           as_retriever(),
                                       return_source_documents
                                           =True,
                                       chain_type_kwargs={"
                                           prompt":
                                           QA_CHAIN_PROMPT})

result = qa_chain({"query": question})
result["result"]
```

Listing 2: Custom Chat Model with LangChain

4.3 Memory

```python

from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True
)
```

Listing 3: Import and Define Memory

4.4 ConversationalRetrievalChain

```python
from langchain.chains import ConversationalRetrievalChain

# Set up the retriever from the vector store
retriever = vectordb.as_retriever()

# Initialize the conversational retrieval chain with memory
qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=memory
)

# Ask a question and get the result
question = "Is probability a class topic?"
result = qa({"question": question})
print(result['answer'])

# Ask another question related to the previous context
question = "Why are those prerequisites needed?"
result = qa({"question": question})
print(result['answer'])
```

Listing 4: Conversational Retrieval Chain Setup

4.5 Create a chatbot that works on your documents

```python

from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter,
    RecursiveCharacterTextSplitter
from langchain.vectorstores import DocArrayInMemorySearch
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA,
    ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import TextLoader
from langchain.document_loaders import PyPDFLoader
```

Listing 5: Import required libraries

4.6 Loading Database and Conversational Retrieval Chain

```python
def load_db(file, chain_type, k):
    # load documents
    loader = PyPDFLoader(file)
    documents = loader.load()
    # split documents
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000, chunk_overlap=150)
    docs = text_splitter.split_documents(documents)
    # define embedding
    embeddings = OpenAIEmbeddings()
    # create vector database from data
    db = DocArrayInMemorySearch.from_documents(docs,
        embeddings)
    # define retriever
    retriever = db.as_retriever(search_type="similarity",
        search_kwargs={"k": k})
    # create a chatbot chain. Memory is managed externally.
    qa = ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(model_name=llm_name, temperature=0),
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa
```

Listing 6: Python Code for Loading Database and Creating a Conversational Retrieval Chain

4.7 Class and Dashboard Implementation

```python
import panel as pn
import param

class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])

    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_file = "docs/cs229_lectures/MachineLearning-Lecture01.pdf"
        self.qa = load_db(self.loaded_file,"stuff", 4)

    def call_load_db(self, count):
        if count == 0 or file_input.value is None:  # init or no file specified :
            return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")
        else:
            file_input.save("temp.pdf")  # local copy
            self.loaded_file = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 4)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded File: {self.loaded_file}")

    def convchain(self, query):
        if not query:
            return pn.WidgetBox(pn.Row('User:', pn.pane.Markdown("", width=600)), scroll=True)
        result = self.qa({"question": query, "chat_history": self.chat_history})
        self.chat_history.extend([(query, result["answer"])])
        self.db_query = result["generated_question"]
        self.db_response = result["source_documents"]
        self.answer = result['answer']
        self.panels.extend([
            pn.Row('User:', pn.pane.Markdown(query, width=600)),
            pn.Row('ChatBot:', pn.pane.Markdown(self.answer, width=600, style={'background-color': '#F6F6F6'}))
        ])
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)

    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )

    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles={'background-color': '#F6F6F6'}))]
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600, scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles={'background-color': '#F6F6F6'}))]
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)

    def clr_history(self,count=0):
        self.chat_history = []
        return
```

Listing 7: Python Code for Class and Dashboard Implementation

4.8 Dashboard

```python
cb = cbfs()

file_input = pn.widgets.FileInput(accept='.pdf')
button_load = pn.widgets.Button(name="Load DB", button_type='primary')
button_clearhistory = pn.widgets.Button(name="Clear History", button_type='warning')
button_clearhistory.on_click(cb.clr_history)
inp = pn.widgets.TextInput( placeholder='Enter text here')

bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)

jpg_pane = pn.pane.Image( './img/convchain.jpg')

tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=300),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
tab3= pn.Column(
    pn.panel(cb.get_chats),
    pn.layout.Divider(),
)
tab4=pn.Column(
    pn.Row( file_input, button_load, bound_button_load),
    pn.Row( button_clearhistory, pn.pane.Markdown("Clears chat history. Can use to start a new topic" )),
    pn.layout.Divider(),
    pn.Row(jpg_pane.clone(width=400))
)
dashboard = pn.Column(
    pn.Row(pn.pane.Markdown('# ChatWithYourData_Bot')),
    pn.Tabs(('Conversation', tab1), ('Database', tab2), ('Chat History', tab3),('Configure', tab4))
)
dashboard
```

Listing 8: Python Code for Dashboard Implementation

5 Explanation Video Link

You can access the explanation video (in Farsi) using the following link.