개발기관 배정연번	2025-XXXX-XX

가. **개발이력**	
나. **최초 개발**	2025.09.01
다. **1차 업데이트**	1.
라. **2차 업데이트**	2.
마. **3차 업데이트**	3.

「AI기반데이터분석및AI Agent

개발과정:

AI기반데이터분석및AI Agent 개발과정」

랭체인 실습

제 출 일 :

소 속 :

성 명 : ※ 워크북 제출시 반드시 파일명을 ○○○○○○(제출년월일6자리)_○○○(성명).zip로 하여 주십시오.(예시: 20251002_홍길동.zip)

※ 제출 메일 : onlooker2zip@naver.com

개발일자: 2025.09 01

간단한 텍스트 생성 체인 만들기

☐ 필요한 모듈 임포트하기

ㅇ 이제 필요한 모듈을 임포트합니다:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
```

☐ 텍스트 생성 체인 만들기

ㅇ LCEL을 사용하면 간단한 파이프라인 형태로 체인을 구성할 수 있습니다. 여기서는 주어진 주제에 대한 짧은 농담을 생성하는 체인을 만들어 보겠습니다:

```
prompt_template = ChatPromptTemplate.from_template("tell me a short joke about {topic}")

chain = prompt_template | ChatOpenAI() | StrOutputParser()
```

ㅇ 이 코드는 다음과 같은 단계로 구성됩니다:

l ChatPromptTemplate.from_template(): 프롬프트 템플릿을 생성합니다. 여기서는 {topic}이라는 변수를 포함한 템플릿을 사용합니다.

l ChatOpenAI(): OpenAI의 채팅 모델을 사용합니다.

l StrOutputParser(): 모델의 출력을 문자열로 파싱합니다.

이 세 단계를 파이프(|) 연산자로 연결하여 하나의 체인을 만듭니다.

☐ 체인 사용하기

ㅇ이제 만든 체인을 사용해 봅시다:

```
result = chain.invoke({'topic': 'cat'})

print(result)
```

이 코드는 'cat'이라는 주제로 짧은 농담을 생성합니다. 결과는 다음과 같을 수 있습니다:

Why was the cat sitting on the computer? Because it wanted to keep an eye on the mouse!

☐ 마치며

ㅇLangChain Expression Language를 사용하면 복잡한 AI 작업을 간단하고 읽기 쉬운 코드로 구현할 수 있습니다. 이 예제에서는 간단한 텍스트 생성 체인을 만들어 보았지만, LCEL의 강력함을 활용하면 훨씬 더 복잡한 애플리케이션도 쉽게 만들 수 있습니다.

텍스트 감성 분류 체인 만들기

☐ 필요한 모듈 임포트하기

ㅇ 먼저, 필요한 라이브러리를 설치해야 합니다. Google Colab을 사용한다면, 다음 명령어로 필요한 패키지를 설치할 수 있습니다:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain.output_parsers.enum import EnumOutputParser
from langchain_openai import ChatOpenAI
from enum import Enum
```

☐ 감성 분류 체인 만들기

ㅇ 이제 본격적으로 감성 분류 체인을 만들어 보겠습니다.

ㅇ 먼저, 감성을 나타내는 열거형(Enum)을 정의합니다:

```
class Sentiment(Enum):
    POSITIVE = 'positive'
    NEGATIVE = 'negative'
```

ㅇ 다음으로, 프롬프트 템플릿을 만듭니다:

```
prompt_template = ChatPromptTemplate.from_template("Classify '{input_text}' as 'positive' or 'negative' only.")
```

ㅇ 마지막으로, 체인을 구성합니다:

```
chain = prompt_template | ChatOpenAI() | EnumOutputParser(enum=Sentiment)
```

ㅇ 이 체인은 다음과 같이 작동합니다:

- prompt_template이 입력 텍스트를 포함한 프롬프트를 생성합니다.
- ChatOpenAI()가 이 프롬프트를 처리하여 응답을 생성합니다.
- EnumOutputParser가 ChatGPT의 응답을 Sentiment 열거형으로 파싱합니다.

☐ 체인 사용하기

ㅇ 이제 만든 체인을 사용해 봅시다:

```
result = chain.invoke({'input_text':'I love LLM.'})
print(result)  # 출력: <Sentiment.POSITIVE: 'positive'>

result = chain.invoke({'input_text':'I hate LLM.'})
print(result)  # 출력: <Sentiment.NEGATIVE: 'negative'>
```

☐ 결과 분석

ㅇ우리의 체인은 간단한 문장에 대해 잘 작동하는 것 같습니다.

```
result = chain.invoke({'input_text':'I love and hate LLM.'})
print(result)  # 출력: <Sentiment.NEGATIVE: 'negative'>

result = chain.invoke({'input_text':'I hate and love LLM.'})
print(result)  # 출력: <Sentiment.NEGATIVE: 'negative'>
```

☐ 마치며

ㅇ이렇게 LangChain을 사용하여 간단한 텍스트 감성 분류 체인을 만들어 보았습니다. 이 예제를 통해 LangChain의 기본적인 사용법과 체인 구성 방법을 배웠습니다.

ㅇ이 모델은 매우 기본적이지만, 더 복잡한 감성 분석이나 다른 종류의 텍스트 분류 작업으로 확장할 수 있습니다.

ㅇLangChain의 강력함은 이런 식으로 다양한 컴포넌트를 쉽게 조합하고 확장할 수 있다는 점입니다.

간단한 챗봇 만들기

☐ 필요한 모듈 임포트하기

ㅇ 이제 필요한 모듈들을 가져와 봅시다:

```
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
```

☐ 챗봇 만들기

ㅇ챗봇을 만드는 과정은 놀랍도록 간단합니다! 다음 코드를 보세요:

```
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "Talk like a close friend and use emojis ��"),
    ("human", "{user_input}")]
)

chain = {"user_input": RunnablePassthrough()} | prompt_template | ChatOpenAI() | StrOutputParser()
```

이 코드는 다음과 같은 일을 합니다:

l 챗봇에게 친한 친구처럼 대화하고 이모지를 사용하라고 지시합니다.

l 사용자의 입력을 받아 처리합니다.

l OpenAI의 ChatGPT 모델을 사용하여 응답을 생성합니다.

l 생성된 응답을 문자열로 변환합니다.

☐ 챗봇과 대화하기

ㅇ 이제 챗봇과 대화를 나눠볼 차례입니다! 다음 코드로 간단한 대화 루프를 만들 수 있습니다:

```
while True:
    user_message = input("USER > ")
    if user_message.lower() == "quit":
        break

    print(" A I > ", end="", flush=True)
    for chunk in chain.stream(user_message):
        print(chunk, end="", flush=True)

    print()
```

이 코드는 사용자의 입력을 받고, 챗봇의 응답을 실시간으로 출력합니다. "quit"을 입력하면 대화가 종료됩니다.

☐ 마무리

ㅇ이 챗봇은 친구처럼 대화하고 이모지를 사용하여 더욱 친근하게 느껴집니다.

수학 연산 체인 구현하기

☐ 준비 단계

○ 먼저, 필요한 라이브러리를 설치하고 임포트해야 합니다.

라이브러리 설치:

```
pip install numexpr
```

☐ 필요한 모듈 임포트

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
import numexpr
```

☐ 체인 구성하기

○ 이제 LangChain을 사용하여 수학 연산 체인을 구성해 보겠습니다.

1. 프롬프트 템플릿 정의:

```
prompt_template = ChatPromptTemplate.from_template(
    "For {query}, write only the mathematical expression suitable for numexpr.evaluate()."
)
```

○ 이 템플릿은 사용자의 쿼리를 받아 numexpr 라이브러리가 평가할 수 있는 수학 표현식으로 변환하도록 ChatGPT에 지시합니다.

2. 체인 구성:

```
chain = prompt_template | ChatOpenAI() | StrOutputParser() | numexpr.evaluate
```

○ 이 한 줄의 코드로 우리는 다음과 같은 강력한 파이프라인을 만들었습니다:

l prompt_template: 사용자 쿼리를 ChatGPT에 적합한 형식으로 변환

l ChatOpenAI(): ChatGPT 모델을 사용하여 수학 표현식 생성

l StrOutputParser(): ChatGPT의 출력을 문자열로 파싱

l numexpr.evaluate: 파싱된 수학 표현식을 실제로 계산

☐ 체인 사용하기

○ 이제 체인을 사용해 봅시다!

```
result = chain.invoke({'query': '4와 5를 더해줘.'})
print(result)
```

○ 자연어로 된 수학 문제를 입력했고, 시스템은 이를 정확하게 해석하여 계산된 결과를 반환했습니다.

ㅇ ChatGPT는 이 쿼리를 해석하여 수학 표현식을 생성합니다.

ㅇ이 표현식은 문자열로 파싱된 후 numexpr.evaluate() 함수에 전달됩니다.

ㅇnumexpr 라이브러리가 이 표현식을 평가하여 최종 결과를 계산합니다.

☐ 결론

ㅇ 이렇게 우리는 LangChain과 ChatGPT를 활용하여 자연어로 된 수학 문제를 해결하는 강력한 도구를 만들었습니다. 이 접근 방식은 단순한 계산을 넘어 복잡한 수학적 개념이나 문제 해결에도 확장하여 적용할 수 있습니다.

PythonREPL을 활용한 동적 코드 실행 챗봇 구현

☐ 필요한 모듈 임포트

ㅇ 이제 필요한 모듈들을 임포트합니다:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_experimental.utilities import PythonREPL
from langchain_core.runnables import RunnablePassthrough
```

☐ 프롬프트 템플릿 설정

ㅇ 챗봇의 동작을 정의하는 프롬프트 템플릿을 설정합니다:

```
template = """
사용자의 문제를 해결하기 위해 파이썬 코드를 작성합니다.
사용자에게 확인이나 추가 정보를 다시 묻지 마세요.
직접 실행할 수 있는 파이썬 코드만 반환하세요.
"""
```

```
prompt_template = ChatPromptTemplate.from_messages(
    [("system", template),
     ("human", "{user_input}")]
)
```

☐ 체인 구성

○ LangChain의 강력한 기능 중 하나는 여러 컴포넌트를 연결하여 복잡한 작업을 수행할 수 있다는 점입니다. 여기서는 다음과 같은 체인을 구성합니다:

```
chain = {"user_input": RunnablePassthrough()} | prompt_template | ChatOpenAI() | StrOutputParser() | PythonREPL().run
```

○ 이 체인은 다음과 같은 과정을 거칩니다:

l 사용자 입력을 받습니다.
l 프롬프트 템플릿에 입력을 적용합니다.
l ChatOpenAI 모델에 전달하여 응답을 생성합니다.
l 응답을 문자열로 파싱합니다.
l 파싱된 응답(Python 코드)을 PythonREPL().run을 통해 실행합니다.

○ 특히 마지막 단계인 PythonREPL().run은 생성된 Python 코드를 실시간으로 실행하는 핵심 기능입니다. 이를 통해 사용자의 요청에 따라 동적으로 코드를 생성하고 즉시 실행할 수 있습니다.

□ 대화 함수 구현

ㅇ사용자와의 대화를 관리하는 함수를 구현합니다:

```
def chat_with_user(user_message):
    max_attempts = 3
    attempts = 0
    while attempts < max_attempts:
        try:
            ai_message = chain.invoke(user_message)
            if "error" in ai_message.lower() or "failed" in ai_message.lower():
                raise Exception(f"Detected error in AI message: {ai_message}")
            return ai_message
        except Exception as e:
            attempts += 1
            print(f"Attempt {attempts}: Error - {e}")
            if attempts == max_attempts:
                return "Sorry, I am unable to process your request at the moment."
```

ㅇ이 함수는 사용자의 메시지를 받아 AI의 응답을 생성하고, PythonREPL을 통해 실행합니다. 오류가 발생하면 최대 3번까지 재시도합니다.

□ 메인 루프

ㅇ마지막으로, 사용자와의 대화를 계속하는 메인 루프를 구현합니다:

```
while True:
    user_message = input("USER > ")
    if user_message.lower() == "quit":
        break
    ai_message = chat_with_user(user_message)
    print(f" A I > {ai_message}")
```

ㅇ이 루프는 사용자로부터 입력을 받고, AI의 응답을 출력합니다. 사용자가 "quit"를 입력하면 프로그램이 종료됩니다.

☐ 결론

ㅇ이렇게 해서 우리는 LangChain과 PythonREPL을 사용하여 동적으로 코드를 실행하는 챗봇을 구현해보았습니다. 이 챗봇은 사용자의 요청에 따라 Python 코드를 생성하고 즉시 실행할 수 있는 강력한 기능을 가지고 있습니다.

결론PythonREPL().run을 사용함으로써, 우리는 단순히 텍스트 응답을 제공하는 것을 넘어서 실제로 작동하는 코드를 생성하고 실행할 수 있게 되었습니다. 이는 프로그래밍 도우미, 자동화 도구, 또는 교육용 애플리케이션 등 다양한 분야에서 활용될 수 있는 강력한 기능입니다.

RAG 체인 구성-PDF 문서 기반 QA챗봇 구현

PDF 문서 로드 및 처리

ㅇ이제 PDF 문서를 로드하고 처리합니다:

```
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma
# PDF 파일이 저장된 폴더 경로를 설정합니다.
folder_path = './datasets/pdfs/'
# 텍스트를 저장할 리스트를 초기화합니다.
texts = [ ]
# 텍스트를 분할할 때 사용할 CharacterTextSplitter 객체를 생성합니다.
text_splitter = CharacterTextSplitter(
    separator = "₩n", # 텍스트를 분할할 때 사용할 구분자
    chunk_size = 1000, # 각 분할된 텍스트의 최대 길이
    chunk_overlap = 50) # 분할된 텍스트 간의 중첩 길이
# 지정된 폴더 내의 모든 파일을 순회합니다.
for filename in os.listdir(folder_path):
    # 파일이 PDF 형식인 경우에만 처리합니다.
    if filename.endswith(".pdf"):
        # PDF 파일을 로드하여 raw_documents에 저장합니다.
        raw_documents = PyPDFLoader(folder_path + '/' + filename).load()
        # 로드된 문서를 분할하여 documents에 저장합니다.
        documents = text_splitter.split_documents(raw_documents)
        # 분할된 문서를 texts 리스트에 추가합니다.
```

```
        texts.extend(documents)
# 분할된 텍스트를 Embeddings로 변환하여 Chroma 데이터베이스에 저장합니다.
db = Chroma.from_documents(texts, OpenAIEmbeddings())
# 데이터베이스에서 검색을 수행할 수 있는 retriever 객체를 생성합니다.
# search_kwargs: 검색 시 사용할 추가 매개변수 (여기서는 상위 10개의 결과를 반환)
retriever = db.as_retriever(search_kwargs={"k": 10})
```

ㅇ이 코드는 PDF 파일을 로드하고, 텍스트를 청크로 나누어 벡터 데이터베이스에 저장합니다.

☐ RAG 체인 구성

ㅇ다음으로, RAG 체인을 구성합니다:

```
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

prompt_template = ChatPromptTemplate.from_template("당신은 질문 답변 작업의 어시스턴트입
    니다. 다음 문맥을 사용하여 질문에 답하세요. 답을 모른다면 모른다고 말하세요. 답변은 최
    대 세 문장으로 작성하고 메타데이터 정보를 포함하여 간결하게 작성하세요. 한국어로 작성합
    니다. ₩ns₩n질문: {question} ₩n문맥: {context} ₩n답변:")

chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt_template
    | ChatOpenAI()
    | StrOutputParser()
)
```

ㅇ이 체인은 사용자의 질문을 받아 관련 문맥을 검색하고, 이를 바탕으로 답변을 생성합니다.

☐ 대화형 인터페이스 구현

ㅇ마지막으로, 간단한 대화형 인터페이스를 만듭니다:

```
def chat_with_user(user_message):
    ai_message = chain.invoke(user_message)
    return ai_message

while True:
    user_message = input("USER > ")
    if user_message.lower() == "quit":
        break
    ai_message = chat_with_user(user_message)
    print(f" A I > {ai_message}")
```

이제 사용자는 질문을 입력하고 시스템은 PDF 문서에서 관련 정보를 찾아 답변을 제공합니다.

☐ 결론

USER > LLM 모델에 대해 설명해 줘.

A I > LLM은 Large Language Models의 약자로, 대규모 데이터셋을 기반으로 사전 훈련된 거대한 언어 모델을 가리킵니다. LLM은 다양한 자연어 처리 작업에 대한 지식을 포괄적으로 보유하고 있으며, 지식 집약적인 작업이나 자연어 이해, 생성 작업 등에서 뛰어난 성과를 보일 수 있습니다. 메타데이터 정보: 페이지 1, 소스: './datasets/pdfs//Harnessing the Power of LLMs in Practice A Survey on ChatGPT and Beyond-2304.13712v2.pdf'.

USER > quit

ㅇ이 RAG 시스템은 대량의 문서에서 정보를 효율적으로 추출하고 질문에 답변할 수 있습니다. 기업 문서, 매뉴얼, 학술 논문 등 다양한 분야에서 활용될 수 있으며, 사용자의 정보 접근성을 크게 향상시킬 수 있습니다.

Memory-대화형 챗봇 만들기

☐ 필요한 모듈 임포트하기

ㅇ이제 필요한 모듈들을 가져올 차례입니다:

```
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain.memory import ConversationBufferWindowMemory
from operator import itemgetter
```

ㅇ각 모듈은 챗봇의 다양한 기능을 구현하는 데 사용됩니다.

☐ 챗봇 설정하기

ㅇ이제 챗봇의 핵심 부분을 설정해 봅시다:

```
# ChatPromptTemplate을 사용하여 대화의 기본 템플릿을 설정합니다.
# 시스템 메시지와 사용자 입력을 포함합니다.
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "Talk like a close friend and use emojis ��"),  # 시스템 메시지: 친근한 친구처
```

```
    럼 말하고 이모지를 사용하세요.
    MessagesPlaceholder(variable_name="chat_history"),  # 대화 기록을 위한 자리 표시자
    ("human", "{user_input}")  # 사용자의 입력을 포함
])

# 대화 메모리를 설정합니다. 최근 3개의 메시지를 반환하도록 설정합니다.
memory = ConversationBufferWindowMemory(k=3, return_messages=True)

# 대화 체인을 설정합니다.
chain = (
    { "user_input": RunnablePassthrough() }  # 사용자의 입력을 그대로 전달
    | RunnablePassthrough.assign(
        chat_history=RunnableLambda(memory.load_memory_variables) | itemgetter("history")
    )  # 메모리에서 대화 기록을 불러와 chat_history 변수에 할당
    | prompt_template  # 설정한 프롬프트 템플릿을 사용
    | ChatOpenAI()  # OpenAI의 챗봇 모델을 사용
    | StrOutputParser()  # 문자열 출력 파서
)
```

여기서 우리는:

- 챗봇의 성격을 정의하는 프롬프트 템플릿을 만들었어요.
- 대화 기억을 관리하는 메모리 객체를 생성했습니다.
- 입력을 처리하고 응답을 생성하는 체인을 구성했어요.

☐ 대화 기능 구현하기

ㅇ 마지막으로, 실제로 사용자와 대화할 수 있는 함수를 만들어 봅시다:

```
def chat_with_user(user_message):
    ai_message = chain.invoke(user_message)
    memory.save_context({"input": user_message}, {"output": ai_message})
    return ai_message

while True:
    user_message = input("USER > ")
    if user_message.lower() == "quit":
        break
    ai_message = chat_with_user(user_message)
    print(f" A I > {ai_message}")
```

ㅇ이 코드는 사용자의 입력을 받아 챗봇의 응답을 생성하고, 대화 내용을 메모리에 저장합니다. 사용자가 "quit"을 입력하면 대화가 종료됩니다.

도구 체인 구축 가이드

☐ 필요한 모듈 임포트하기

ㅇ먼저, 필요한 라이브러리를 설치하고 임포트합니다.

```
import json
import requests
from langchain_core.tools import tool
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
from langchain.schema import AIMessage
```

☐ 도구 만들기

ㅇ이제 우리의 첫 번째 도구인 날씨 정보 가져오기 도구를 만들어 보겠습니다.

```
@tool
def get_current_weather(latitude: float, longitude: float) -> str:
    """
```

```
    Open-Meteo API를 사용하여 지정된 위도와 경도의 현재 날씨 데이터를 가져옵니다.
    """
    base_url = "https://api.open-meteo.com/v1/forecast"
    params = {
        "latitude": latitude,
        "longitude": longitude,
        "current_weather": True,
        "daily": "temperature_2m_max,temperature_2m_min",
        "timezone": "auto"
    }
    response = requests.get(base_url, params=params)
    info = {}
    if response.status_code == 200:
        info = response.json()
    return json.dumps(info)
```

ㅇ이 도구는 Open-Meteo API를 사용하여 주어진 위도와 경도의 현재 날씨 정보를 가져옵니다.

ㅇ다음으로, 현재 위치를 가져오는 도구를 만들어 보겠습니다.

```
@tool
def get_current_location() -> str:
    """
    IPinfo.io API를 사용하여 IP 주소 기반의 현재 위치를 가져옵니다.
    """
    response = requests.get('https://ipinfo.io')
    info = response.json()
    return json.dumps(info)
```

ㅇ이 도구는 IPinfo.io API를 사용하여 현재 IP 주소 기반의 위치 정보를 가져옵니다.

☐ LLM 설정

ㅇ도구를 사용할 LLM(Large Language Model)을 설정해 보겠습니다.

```
llm = ChatOpenAI(model="gpt-3.5-turbo")
tools = [get_current_weather, get_current_location]
llm_with_tools = llm.bind_tools(tools)
```

ㅇ여기서 우리는 OpenAI의 GPT-3.5-turbo 모델을 사용하고, 우리가 만든 도구들을 이 모델에 바인딩했습니다.

☐ 도구 호출 헬퍼 함수 만들기

ㅇ다음으로, 도구를 순차적으로 호출하는 헬퍼 함수를 만들어 보겠습니다.

```
def call_tools(msg: AIMessage):
    """순차적인 도구 호출을 위한 간단한 헬퍼 함수입니다."""
    tool_map = {tool.name: tool for tool in tools}
    tool_calls = msg.tool_calls.copy()
    for tool_call in tool_calls:
        tool_call["output"] = tool_map[tool_call["name"]].invoke(tool_call["args"])
    return tool_calls
```

이 함수는 AI 메시지에 포함된 도구 호출을 순차적으로 실행하고 그 결과를 반환합니다.

☐ 체인 구성

ㅇ이제 우리의 LLM과 도구 호출 헬퍼 함수를 연결하여 체인을 구성해 보겠습니다.

```
chain = llm_with_tools | call_tools
```

ㅇ이 체인은 사용자의 입력을 받아 LLM이 처리한 후, 필요한 도구를 호출하고 그 결과를 반환합니다.

☐ 체인 사용하기

ㅇ이제 우리가 만든 체인을 사용해 볼 차례입니다!

```
result = chain.invoke("What is the weather in Seoul?")
print(result)
result = chain.invoke("What is the current location?")
print(result)
```

이 코드는 서울의 날씨를 물어보고, 현재 위치를 확인하는 쿼리를 실행합니다.

☐ 대화형 인터페이스 만들기

ㅇ마지막으로, 사용자와 대화할 수 있는 간단한 인터페이스를 만들어 보겠습니다.

```
def chat_with_user(user_message):
    ai_message = chain.invoke(user_message)
    return ai_message
while True:
    user_message = input("USER > ")
    if user_message.lower() == "quit":
        break
    ai_message = chat_with_user(user_message)
    print(f" A I > {ai_message}")
```

ㅇ이 코드는 사용자의 입력을 받아 우리의 체인을 통해 처리하고, 그 결과를 출력합니다. 'quit'를 입력하면 대화가 종료됩니다.

□ 결론

ㅇ이렇게 해서 우리는 LangChain을 사용하여 날씨 정보와 위치 정보를 가져올 수 있는 도구 체인을 구축해 보았습니다. 이 체인은 사용자의 질문에 답하기 위해 필요한 도구를 자동으로 선택하고 실행합니다.

라우팅 체인 - 유연한 AI 응답 시스템 구축하기

☐ 필요한 모듈 임포트하기

ㅇ

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate
```

☐ 간단한 분류 체인 만들기

ㅇ우리의 첫 번째 단계는 사용자의 질문을 'LangChain', 'OpenAI', 또는 'Other' 중 하나로 분류하는 간단한 체인을 만드는 것입니다.

```
chain = (
    PromptTemplate.from_template(
        """아래의 사용자 질문이 주어졌을 때, 'LangChain', 'OpenAI', 'Other' 중 어느 것에 관한 질문인지 분류하세요.

두 단어 이상으로 응답하지 마세요.

질문: {question}
분류:"""
```

```
    )
    | ChatOpenAI()
    | StrOutputParser()
)

result = chain.invoke({
    "question": "OpenAI를 호출하려면 어떻게 하나요?"
})
print(result)  # 출력: OpenAI
```

이 체인은 사용자의 질문을 입력받아 한 단어로 분류합니다.

☐ 특화된 응답 체인 만들기

ㅇ다음으로, 각 카테고리에 대한 특화된 응답 체인을 만듭니다.

```
langchain_chain = PromptTemplate.from_template(
    """귀하는 랭체인 전문가입니다.
항상 "해리슨 체이스가 말했듯이"로 시작하는 질문에 답하세요.
다음 질문에 답하세요:

질문: {question}
답변:"""
)  | ChatOpenAI()

openai_chain = PromptTemplate.from_template(
    """귀하는 OpenAI 전문가입니다.
항상 "사무엘 해리스 알트먼이 나에게 말했듯이"로 시작하는 질문에 답하세요.
다음 질문에 답하세요:
```

```
질문: {question}
답변:"""
) | ChatOpenAI()

general_chain = PromptTemplate.from_template(
    """다음 질문에 답변하세요:

질문: {question}
답변:"""
) | ChatOpenAI()
```

각 체인은 특정 주제에 대해 전문가처럼 대답하도록 설계되었습니다.

☐ 라우팅 함수 정의

ㅇ이제 분류 결과에 따라 적절한 체인을 선택하는 라우팅 함수를 정의합니다.

```
def route(info):
    if "openai" in info["topic"].lower():
        return openai_chain
    elif "langchain" in info["topic"].lower():
        return langchain_chain
    else:
        return general_chain
```

☐ 전체 체인 조립하기

ㅇ마지막으로, 모든 구성 요소를 하나의 체인으로 조립합니다.

```
from langchain_core.runnables import RunnableLambda

full_chain = {
    "topic": chain,
    "question": lambda x: x["question"]
} | RunnableLambda(route)
```

이 체인은 입력된 질문을 분류하고, 그 결과에 따라 적절한 응답 체인으로 라우팅합니다.

☐ 체인 사용하기

ㅇ이제 우리의 체인을 다양한 질문으로 테스트해 볼 수 있습니다.

```
print(full_chain.invoke({"question": "OpenAI는 어떻게 사용하나요?"}))
print(full_chain.invoke({"question": "랭체인은 어떻게 사용하나요?"}))
print(full_chain.invoke({"question": "2 + 2는?"}))
```

각 질문은 적절한 전문가 체인으로 라우팅되어 답변됩니다.

☐ 고급 라우팅: 임베딩 기반 프롬프트 선택

ㅇ더 복잡한 시나리오에서는 코사인 유사도를 사용하여 가장 적절한 프롬프트를 선택할 수 있습니다.

```
from langchain.utils.math import cosine_similarity
from langchain_openai import OpenAIEmbeddings

# 물리학 질문에 대한 템플릿을 정의합니다.
physics_template = """당신은 매우 똑똑한 물리학 교수입니다.
물리학에 대한 질문에 간결하고 이해하기 쉽게 대답하는 데 능숙합니다.
질문에 대한 답을 모를 때는 모른다고 인정합니다.

다음은 질문입니다:
{query}"""

# 수학 질문에 대한 템플릿을 정의합니다.
math_template = """당신은 아주 훌륭한 수학자입니다. 당신은 수학 문제에 대한 답을 잘합니다.
당신은 어려운 문제를 구성 요소로 분해하고, "구성 요소에 답한 다음 "더 넓은 질문에 답할 수 있기 때문에 매우 훌륭합니다.
구성 요소에 답한 다음 이를 종합하여 더 넓은 질문에 답할 수 있기 때문입니다.

여기 질문이 있습니다:
{query}"""

# OpenAIEmbeddings 객체를 생성합니다.
embeddings = OpenAIEmbeddings()

# 물리학과 수학 템플릿을 리스트로 묶습니다.
prompt_templates = [physics_template, math_template]

# 템플릿들을 임베딩합니다.
prompt_embeddings = embeddings.embed_documents(prompt_templates)
```

```
# 입력된 질문에 따라 적절한 템플릿을 선택하는 함수입니다.
def prompt_router(input):
    # 입력된 질문을 임베딩합니다.
    query_embedding = embeddings.embed_query(input["query"])

    # 질문 임베딩과 템플릿 임베딩 간의 코사인 유사도를 계산합니다.
    similarity = cosine_similarity([query_embedding], prompt_embeddings)[0]

    # 가장 유사한 템플릿을 선택합니다.
    most_similar = prompt_templates[similarity.argmax()]

    # 선택된 템플릿이 수학 템플릿인지 물리학 템플릿인지 출력합니다.
    print("Using MATH" if most_similar == math_template else "Using PHYSICS")

    # 선택된 템플릿을 반환합니다.
    return PromptTemplate.from_template(most_similar)


# 체인을 구성합니다.
chain = (
    { "query": RunnablePassthrough() }  # 입력된 질문을 그대로 통과시킵니다.
    | RunnableLambda(prompt_router)  # 질문에 따라 적절한 템플릿을 선택합니다.
    | ChatOpenAI()  # 선택된 템플릿을 사용하여 OpenAI와 대화합니다.
    | StrOutputParser()  # 결과를 문자열로 파싱합니다.
)

# 체인을 실행하여 질문에 대한 답변을 출력합니다.
print(chain.invoke("블랙홀이란 무엇인가요?"))
print(chain.invoke("경로 통합이란 무엇인가요?"))
```

이 고급 라우팅 방식은 질문의 내용을 더 깊이 이해하고, 가장 적합한 전문가 프롬프트를 선택합

니다.

□ 결론

ㅇ LangChain의 라우팅 체인을 사용하면 다양한 질문에 대해 맥락에 맞는 정확한 응답을 제공할 수 있는 유연한 AI 시스템을 구축할 수 있습니다. 이는 chatbot, 질문-답변 시스템, 또는 지식 기반 애플리케이션을 개발할 때 매우 유용한 기능입니다.

RAG—대학교 학생 성적에 영향을 미치는 요소를 탐색하고 분석하는 시스템 구축

"대학교 학생 성적에 영향을 주는 요소"를 찾는 예제를 RAG로 구성합니다. 이 예제에서는 RAG 시스템이 성적에 영향을 미치는 다양한 요소를 조사하고, 이를 통해 성적 향상에 대한 제안을 생성하는 방식으로 설계될 수 있습니다.

RAG 구성 예제: 대학교 학생 성적에 영향을 미치는 요소 찾기

1. 목표

- 대학교 학생들의 성적에 영향을 미치는 다양한 요소들을 데이터베이스나 학습 자료에서 검색하고, 이를 바탕으로 분석하여 성적 향상을 위한 조언을 제공하는 시스템을 구축합니다.

2. RAG 구성 흐름

RAG 시스템은 다음과 같은 두 가지 주요 부분으로 구성됩니다:

1. 검색(retrieval): 관련 자료를 외부에서 가져옵니다. (성적과 관련된 연구 자료, 논문, 데이터베이스)
2. 생성(generation): 검색된 자료를 바탕으로, AI가 답변을 생성합니다.

3. 예시 구현

Python의 langchain 라이브러리와 함께 GPT-4를 사용하는 방식으로 예제를 구현할 수 있습니다. 검색 부분에서는 성적에 영향을 미치는 요인들을 검색하고, 이를 바탕으로 성적 개선을 위한 조언을 생성합니다.

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.document_loaders import DirectoryLoader
from langchain.document_loaders import PyMuPDFLoader  # PDF 로더 추가
from langchain.chains import StuffDocumentsChain
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# 1. 데이터 로드: 외부 데이터로 학습 자료 로드
loader = DirectoryLoader(
    './student_grades_research_papers',
    glob="**/*.pdf",  # PDF 파일만 로드
    loader_cls=PyMuPDFLoader  # PDF 로더 사용
)documents = loader.load()
# 문서 내용 확인 (예: 5번째 문서의 첫 300자 출력)
if len(documents) > 4:
    print(documents[4].page_content[:300])

# 2. 문서 임베딩: 문서 임베딩을 생성하여 검색이 가능하도록 설정
```

```
embeddings = OpenAIEmbeddings()
vector_store = FAISS.from_documents(documents, embeddings)

# 3. 검색 쿼리 설정
retriever = vector_store.as_retriever()

# 4. 검색과 생성 모델을 연결하는 RetrievalQA 구성
llm = OpenAI(temperature=0.5)
qa_chain = RetrievalQA(llm=llm, retriever=retriever)

# 5. Prompt 설정
prompt = """
대학교 학생의 성적에 영향을 주는 요소들을 설명해 줘.
검색된 연구 자료를 바탕으로 분석해 주고, 성적 향상에 도움이 될 수 있는 전략을 제시해 줘.
"""

# 6. 질문 실행
result = qa_chain.run(prompt)
print(result)
```

코드 설명

1. 데이터 로드: SimpleDirectoryLoader는 학생 성적과 관련된 연구 자료나 논문을

로드하는 역할을 합니다. 이 예제에서는 './student_grades_research_papers'라는 폴더에 연구 논문이 저장되어 있다고 가정합니다.

2. 문서 임베딩: 문서의 임베딩을 생성하여 나중에 검색할 수 있도록 FAISS를 사용하여 벡터 저장소에 저장합니다. 이 임베딩은 검색 요청에 맞는 문서를 찾기 위해 사용됩니다.

3. 검색 쿼리 설정: vector_store로부터 관련 문서를 검색할 수 있는 retriever를 생성합니다.

4. LLM 연결: GPT-4와 같은 대규모 언어 모델(LLM)을 사용하여 검색된 자료에 기반해 텍스트를 생성합니다.

5. Prompt 설정: 학생 성적에 영향을 미치는 요소를 분석하고, 성적 향상을 위한 전략을 제시하는 요청을 설정합니다.

6. 결과 출력: 검색된 자료와 생성 모델을 통해 최종 분석 결과를 생성합니다.

예시 응답

대학생 성적에 영향을 미치는 요소로는 여러 가지가 있습니다:

1. **공부 습관**: 규칙적인 공부 습관과 시간 관리가 높은 성적과 밀접하게 연관되어 있습니다.

2. **출석률**: 연구에 따르면 높은 출석률이 성적 향상에 긍정적인 영향을 미칩니다.

3. **수면 시간**: 충분한 수면을 취한 학생이 더 좋은 성적을 받을 가능성이 높습니다.

4. **사회적 지원**: 친구나 가족으로부터의 정서적 지지 또한 성적에 긍정적인 영향을 줍니다.

5. **교수와의 상호작용**: 학생들이 교수와 자주 상호작용할수록 성적이 향상되는 경향이

있습니다.

성적 향상을 위한 전략으로는 다음을 제안합니다:

- 하루 최소 2시간씩 꾸준한 공부 시간을 확보하고, 복습 시간을 계획하세요.
- 출석을 철저히 지키고, 강의 중 적극적으로 질문을 하세요.
- 충분한 수면을 유지하고, 피로를 피하기 위해 하루에 7~8시간 수면을 권장합니다.
- 친구와 스터디 그룹을 만들어 협력 학습을 시도하세요.

이 예시에서는 검색된 데이터를 기반으로 분석하고, 성적을 향상시킬 수 있는 여러 가지 전략을 제시하는 방식으로 결과를 생성했습니다.

RAG의 장점

- 외부 데이터 검색을 통해 더 풍부하고 신뢰할 수 있는 정보를 제공할 수 있습니다.
- GPT와 같은 생성 모델의 능력을 활용하여 사용자가 이해하기 쉽게 텍스트를 작성합니다.

이러한 방식으로 대학교 학생 성적에 영향을 미치는 요소를 탐색하고 분석하는 시스템을 구축할 수 있습니다.