

LangChain 기초



참고 자료

- 위키독스 : **랭체인(LangChain) 입문부터 응용까지** [<https://wikidocs.net/edit/book/14473>] (<https://wikidocs.net/edit/book/14473>)
- 유튜브 : **랭체인(LangChain) 강의** [<https://www.youtube.com/channel/UCh-c-LFH9Q6VbHRRqD4oL0A>] (<https://www.youtube.com/channel/UCh-c-LFH9Q6VbHRRqD4oL0A>)

코드

[001_langchain_basics.ipynb](#)

[002_langchain_RAG.ipynb](#)

[LCEL.ipynb](#)

기본 체인 (Prompt + LLM)

- LLM 기반 애플리케이션에서 핵심 개념 중 하나
- 사용자의 입력(프롬프트)을 받아 LLM을 통해 적절한 응답이나 결과를 생성하는 구조
- 대화형 AI, 자동 문서 생성, 데이터 분석 및 요약 등 다양한 용도로 활용 가능

- 구성요소
 - 프롬프트(Prompt)
 - 사용자 또는 시스템에서 제공하는 입력.
 - LLM에게 특정 작업을 수행하도록 요청하는 지시문
 - LLM(Large Language Model)
 - GPT, Gemini 등 대규모 언어 모델
 - 대량의 텍스트 데이터에서 학습해 언어를 이해하고 생성할 수 있는 인공지능 시스템
 - 프롬프트를 바탕으로 적절한 응답을 생성하거나, 주어진 작업을 수행하는 데 사용

일반적인 작동 방식

1. 프롬프트 생성

- 사용자 요구 사항이나 특정 작업을 정의하는 프롬프트 생성
- LLM에게 전달되기 전에, 작업의 목적과 맥락을 명확하게 전달하기 위해 최적화 가능

2. LLM 처리

- 제공된 프롬프트를 분석하고, 학습된 지식을 바탕으로 적절한 응답 생성
- LLM은 내부적으로 다양한 언어 패턴과 지식을 활용해 요청된 작업을 수행하거나 정보 제공

3. 응답 반환

- 생성된 응답은 애플리케이션으로 반환되며, 최종 사용자에게 제공됨
- 직접적인 답변, 생성된 텍스트, 요약된 정보 등 다양한 형태를 취함

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

# prompt + model + output parser
prompt = ChatPromptTemplate.from_template("You are an exper
```

```

t in astronomy. Answer the question. <Question>: {input}")
llm = ChatOpenAI(model="gpt-3.5-turbo-0125")
output_parser = StrOutputParser()

# LCEL chaining
chain = prompt | llm | output_parser

# chain 호출
chain.invoke({"input": "지구의 자전 주기는?"})

```

멀티 체인 (Multi-Chain)

- 여러 개의 체인을 연결하거나 복합적으로 작용하는 것은 멀티 체인 구조를 통해 이루어짐
- 이러한 구조는 각기 다른 목적을 가진 여러 체인을 조합해 입력 데이터를 다양한 방식으로 처리하고 최종적인 결과를 도출할 수 있게 함
- 복잡한 데이터 처리, 의사 결정, AI 기반 작업 흐름 설계 시, 유용

1. 순차적인 체인 연결

- 2개의 체인을 정의하고, 순차적으로 연결해 수행
- Chain 1에서 영어 단어를 입력받음
- Chain2에서 번역된 단어를 변수에 저장한 후, 두 번째 체인의 입력으로 제공
- Chain2에서 전달된 단어를 한국어로 설명하는 작업 수행
- 최종 출력 확인

```

prompt1 = ChatPromptTemplate.from_template("translates {korean_word} to English.")
prompt2 = ChatPromptTemplate.from_template(
    "explain {english_word} using oxford dictionary to me in Korean."
)

llm = ChatOpenAI(model="gpt-3.5-turbo-0125")

```

```

chain1 = prompt1 | llm | StrOutputParser()

chain1.invoke({"korean_word": "미래"})

chain2 = (
    {"english_word": chain1}
    | prompt2
    | llm
    | StrOutputParser()
)

chain2.invoke({"korean_word": "미래"})

```

프롬프트

- 사용자와 언어 모델 간 대화에서 질문이나 요청의 형태로 제시되는 입력문
- 모델이 어떤 유형의 응답을 제공할지 결정하는 데 중요한 역할을 함
- 프롬프트 템플릿 (Prompt Template)
 - 단일 문장 또는 간단한 명령을 입력해 단일 문장 또는 간단한 응답을 생성하는 데 사용되는 프롬프트를 구성할 수 있는 문자열 템플릿
 - Python의 문자열 포매팅을 사용해 동적으로 특정 위치에 입력 값 포함 가능

```

from langchain_core.prompts import PromptTemplate

# 'name'과 'age'라는 두 개의 변수를 사용하는 프롬프트 템플릿을 정의
template_text = "안녕하세요, 제 이름은 {name}이고, 나이는 {age}살입니다."

# PromptTemplate 인스턴스를 생성
prompt_template = PromptTemplate.from_template(template_text)

# 템플릿에 값을 채워서 프롬프트를 완성

```

```
filled_prompt = prompt_template.format(name="홍길동", age=30)
```

```
filled_prompt
```

```
# 문자열 템플릿 결합 (PromptTemplate + PromptTemplate + 문자열)
```

```
combined_prompt = (
    prompt_template
    + PromptTemplate.from_template("\n\n아버지를 아버지라 부를 수 없습니다.")
    + "\n\n{language}로 번역해주세요."
)
```

```
combined_prompt
```

```
combined_prompt.format(name="홍길동", age=30, language="영어")
```

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

```
llm = ChatOpenAI(model="gpt-3.5-turbo-0125")
chain = combined_prompt | llm | StrOutputParser()
chain.invoke({"age":30, "language":"영어", "name":"홍길동"})
```

- LLM의 역할을 정의해 질의 생성하기

```
# 2-튜플 형태의 메시지 목록으로 프롬프트 생성 (type, content)
```

```
from langchain_core.prompts import ChatPromptTemplate
```

```
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "이 시스템은 천문학 질문에 답변할 수 있습니다."),
    ("user", "{user_input}"),
])
```

```

])

messages = chat_prompt.format_messages(user_input="태양계
에서 가장 큰 행성은 무엇인가요?")
messages

chain = chat_prompt | llm | StrOutputParser()

chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인
가요?"})

```

MessagePromptTemplate 활용

```

from langchain_core.prompts import SystemMessagePromptTe
mplate, HumanMessagePromptTemplate

chat_prompt = ChatPromptTemplate.from_messages(
    [
        SystemMessagePromptTemplate.from_template("이 시
스템은 천문학 질문에 답변할 수 있습니다."),
        HumanMessagePromptTemplate.from_template("{user_
input}"),
    ]
)

messages = chat_prompt.format_messages(user_input="태양계
에서 가장 큰 행성은 무엇인가요?")
messages

chain = chat_prompt | llm | StrOutputParser()

chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인
가요?"})

```

LangChain 모델 유형

- LLM과 Chat Model 클래스는 각각 다른 형태의 입력과 출력을 다루는 언어 모델을 나타냄
- 각기 다른 특성과 용도를 가지고 있어, 요구사항에 맞게 선택해 사용
- LLM은 주로 단일 요청에 대한 복잡한 출력에 적합
- Chat Model은 사용자와의 상호작용을 통한 채팅 유형에 적합

LLM

- 텍스트 문자열을 입력으로 받아 처리한 후, 텍스트 문자열 반환
- 광범위한 언어 이해 및 텍스트 생성 작업에 사용
- 문서 요약, 콘텐츠 생성, 질문에 대한 답변 생성 등 복잡한 자연어 처리 작업 수행

```
from langchain_openai import OpenAI

llm = OpenAI()

llm.invoke("한국의 대표적인 관광지 3군데를 추천해주세요.")
```

Chat Model

- 메시지의 리스트를 입력으로 받고, 하나의 메시지를 반환
- 대화형 상황에 최적화
- 사용자와의 연속적인 대화를 처리하는 데 사용
- 대화의 맥락을 유지하면서 적절한 응답을 생성하는 데 중점

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

chat = ChatOpenAI()
# system을 통해 LLM의 역할 정의
chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "이 시스템은 여행 전문가입니다."),
    ("user", "{user_input}"),
```

```
] )
```

```
chain = chat_prompt | chat
chain.invoke({"user_input": "안녕하세요? 한국의 대표적인 관광지
3군데를 추천해주세요."})
```

LangChain의 LLM 모델 파라미터 설정

- LLM 모델의 기본 속성값을 조정하는 방법
- 모델의 속성에 해당하는 모델 파라미터는 LLM의 출력을 조정하고 최적화하는데 사용되며, 모델이 생성하는 텍스트의 스타일, 길이, 정확도 등에 영향을 줌
- 사용하는 모델이나 플랫폼에 따라 다름
- 모델 생성 단계나 모델 호출 단계에서 설정 가능
- bind를 통해 새로운 모델을 불러와 체인에 추가해줄 수 있음
- 주요 파라미터
 - Temperature
 - 생성된 텍스트의 다양성 조정
 - 값이 작으면 예측 가능하고 일관된 출력
 - 값이 크면 다양하고 예측하기 어려운 출력 생성
 - Max Tokens
 - 생성할 최대 토큰 수 지정
 - 생성할 텍스트의 길이 제한
 - Top P
 - 생성 과정에서 특정 확률 분포 내에서 상위 P% 토큰만을 고려
 - 출력의 다양성 조정
 - Frequency Penalty
 - 값이 클수록 이미 등장한 단어나 구절이 다시 등장할 확률 감소시킴
 - 반복을 줄이고 텍스트의 다양성 증가시킴
 - Presence Penalty

- 텍스트 내에서 단어의 존재 유무에 따라 그 단어의 선택 확률 조정
- 값이 클수록 아직 텍스트에 등장하지 않은 새로운 단어 사용 장려
- Stop Sequences
 - 특정 단어나 구절이 등장할 경우 생성 멈추도록 설정
 - 출력을 특정 포인트에서 종료하고자 할 때 사용

```
from langchain_openai import ChatOpenAI

# 모델 파라미터 설정
params = {
    "temperature": 0.7,          # 생성된 텍스트의 다양성 조정
    "max_tokens": 100,          # 생성할 최대 토큰 수
}

kwargs = {
    "frequency_penalty": 0.5,    # 이미 등장한 단어의 재등장 확률
    "presence_penalty": 0.5,     # 새로운 단어의 도입을 장려
    "stop": ["\n"]              # 정지 시퀀스 설정
}

# 모델 인스턴스를 생성할 때 설정
model = ChatOpenAI(model="gpt-3.5-turbo-0125", **params, model_kwargs = kwargs)

# 모델 호출
question = "태양계에서 가장 큰 행성은 무엇인가요?"
response = model.invoke(input=question)

# 전체 응답 출력
print(response)
```

```
from langchain_core.prompts import ChatPromptTemplate
```

```

prompt = ChatPromptTemplate.from_messages([
    ("system", "이 시스템은 천문학 질문에 답변할 수 있습니다."),
    ("user", "{user_input}"),
])

model = ChatOpenAI(model="gpt-3.5-turbo-0125", max_tokens=100)

messages = prompt.format_messages(user_input="태양계에서 가장 큰 행성은 무엇인가요?")

before_answer = model.invoke(messages)

# # binding 이전 출력
print(before_answer)

# 모델 호출 시 추가적인 인수를 전달하기 위해 bind 메서드 사용 (응답의 최대 길이를 10 토큰으로 제한)
chain = prompt | model.bind(max_tokens=10)

after_answer = chain.invoke({"user_input": "태양계에서 가장 큰 행성은 무엇인가요?"})

# binding 이후 출력
print(after_answer)

```

RAG (Retrieval - Augmented Generation) 기본 개념과 구성 요소

- 기존 대규모 언어 모델(LLM)을 확장해 주어진 컨텍스트나 질문에 대해 더욱 정확하고 풍부한 정보를 제공하는 방법
- 학습 데이터에 포함되지 않은 외부 데이터를 실시간으로 검색(Retrieval)하고, 이를 바탕으로 답변을 생성(Generation)하는 과정 포함
- 특히 환각(생성 내용이 사실이 아닌 것으로 오인되는 현상)을 방지

- 모델이 최신 정보를 반영하거나 더 넓은 지식을 활용할 수 있게 함

RAG 모델의 기본 구조

1. 검색 단계(Retrieval Phase)

- 사용자 질문이나 컨텍스트를 입력으로 받아서, 이와 관련된 외부 데이터를 검색하는 단계
- 검색 엔진이나 DB(질문과 관련된 데이터들을 Vector 형태로 임베딩 후, 이 벡터 자체를 저장하는 Vector Store) 등 다양한 소스에서 필요한 정보를 찾아냄
- 검색된 데이터는 질문에 대한 답변을 생성하는데 적합하고 상세한 정보를 포함하는 것을 목표로 함

2. 생성 단계(Generation Phase)

- 검색된 데이터를 기반으로 LLM 모델이 사용자 질문에 답변을 생성하는 단계
- 모델은 검색된 정보와 기존 지식을 결합해 주어진 질문에 답변 생성

RAG 모델의 장점

- 풍부한 정보 제공
 - 검색을 통해 얻은 외부 데이터를 활용해 보다 구체적이고 풍부한 정보 제공
- 실시간 정보 반영
 - 최신 데이터를 검색해 반영함으로써, 모델이 실시간으로 변화하는 정보에 대응 가능
- 환각 방지
 - 검색을 통해 실제 데이터에 기반한 답변을 생성함으로써, 환각 현상이 발생할 위험을 줄이고 정확도를 높일 수 있음

과정 : Load Data - Text Spilt - Indexing - Retrieval - Generation

1. Load Data

```
# Data Loader - 웹 페이지 데이터 가져오기
from langchain_community.document_loaders import WebBase
Loader

# 위키피디아 정책과 지침
url = 'https://ko.wikipedia.org/wiki/%EC%9C%84%ED%82%A4%
EB%B0%B1%EA%B3%BC:%EC%A0%95%EC%B1%85%EA%B3%BC_%EC%A7%80%
EC%B9%A8'
loader = WebBaseLoader(url)

# 웹페이지 텍스트 -> Documents
docs = loader.load()

print(len(docs))
print(len(docs[0].page_content))
print(docs[0].page_content[5000:6000])
```

2. Split Texts

```
# Text Split (Documents -> small chunks: Documents)
from langchain.text_splitter import RecursiveCharacterTe
xtSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunks_overlap=200)
splits = text_splitter.split_documents(docs)

print(len(splits))
print(splits[10])
```

3. Indexing

```
# Indexing (Texts -> Embedding -> Store)
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

# Embedding 모델을 적용할 것인지 embedding에서 지정
vectorstore = Chroma.from_documents(documents=splits, em
```

```

bedding = OpenAIEmbeddings()

docs = vectorstore.similarity_search("격하 과정에 대해서 설명해주세요.")
print(len(docs))
print(docs[0].page_content)

```

4. Retrieval ~ Generation

```

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

# Prompt
template = '''Answer the question based only on the following context:
[context]

Question: [question]
'''

prompt = ChatPromptTemplate.from_template(template)

# LLM
model = ChatOpenAI(model = 'gpt-3.5-turbo-0125', temperature = 0)

# Retriever
retriever = vectorstore.as_retriever()

# Combine Documents
def format_docs(docs):
    return '\n\n'.join(doc.page_content for doc in docs)

# RAG Chain 연결
rag_chain = (

```

```
        {'context': retriever | format_docs, 'question': RunnablePassthrough()}
        | prompt
        | model
        | StrOutputParser()
    )

# Chain 실행
rag_chain.invoke("격하 과정에 대해서 설명해주세요.")
```

LCEL (LangChain Expression Language)

LangChain Expression Language