DATA
SCIENCE LAB

Retrieval Augmented Generation (RAG) with LangChain

24.08.13 / 11기 김지원

CONTENTS

Part 1.

Introduction to LangChain

1. LangChain이란?

LangChain 이란?

랭체인(LangChain): 대규모 언어 모델을 활용한 혁신적인 프레임워크

개발자들이 챗봇, 질의응답 시스템, 자동 요약 등 다양한 LLM 애플리케이션을 쉽게 개발할 수 있도록 지원하는 프레임워크를 만들었습니다.

1. LangChain이란?

LangChain 프레임워크의 구성

랭체인(LangChain) 프레임워크는 LLM 애플리케이션 개발에 도움이 되는 여러 구성 요소로 이루어져 있습니다. 특히 개발자들이 다양한 LLM 작업을 신속하게 구축하고 배포할 수 있도록 설계되었습니다. 랭체인의 주요 구성 요소는 다음과 같습니다.

1. LangChain이란?

LangChain 프레임워크의 구성

랭체인 라이브러리(LangChain Libraries): 파이썬과 자바스크립트 라이브러리를 포함하며, 다양한 컴포넌트의 인터페이스와 통합, 이 컴포넌트들을 체인과 에이전트로 결합할 수 있는 기본 런타임, 그리고 체인과 에이전트의 사용 가능한 구현이 가능합니다.

랭체인 템플릿(LangChain Templates): 다양한 작업을 위한 쉽게 배포할 수 있는 참조 아키텍처 모음입니다. 이 템플릿은 개발자들이 특정 작업에 맞춰 빠르게 애플리케이션을 구축할 수 있도록 돕습니다.

랭서브(LangServe): 랭체인 체인을 REST API로 배포할 수 있게 하는 라이브러리입니다. 이를 통해 개발자들은 자신의 애플리케이션을 외부 시스템과 쉽게 통합할 수 있습니다.

랭스미스(LangSmith): 개발자 플랫폼으로, LLM 프레임워크에서 구축된 체인을 디버깅, 테스트, 평가, 모니터링할 수 있으며, 랭체인과의 원활한 통합을 지원합니다.

1. LangChain이란?

필수 라이브러리 설치

OpenAI에서 제공하는 LLM을 사용하려면 langchain-openai 의존성 라이브러리를 설치해야 합니다.

```
pip install langchain
```

OpenAI 인증키 등록

```
import os
os.environ['OPENAI_API_KEY'] = 'OPENAI_API_KEY'
```

*API 키는 민감한 정보이므로 안전하게 관리하고 노출되지 않도록 주의해야 합니다.

2. Basic LLM Chain

랭체인의 체인(Chain) 개념을 이해하고, 가장 단순한 형태의 기본 LLM 체인부터 체인을 연결하는 멀티 체인 개념까지 다룹니다.

기본 LLM 체인 (Prompt + LLM)

기본 LLM 체인(Prompt + LLM)은 LLM 기반 애플리케이션 개발에서 핵심적인 개념 중 하나입니다. 이러한 체인은 사용자의 입력(프롬프트)을 받아 LLM을 통해 적절한 응답이나 결과를 생성하는 구조를 말합니다. 이 과정은 대화형 AI, 자동 문서 생성, 데이터 분석 및 요약 등 다양한 용도로 활용될 수 있습니다.

2. Basic LLM Chain

기본 LLM 체인 (Prompt + LLM)

1) 기본 LLM 체인의 구성 요소

- 프롬프트(Prompt): 사용자 또는 시스템에서 제공하는 입력으로, LLM에게 특정 작업을 수행하도록 요청하는 지시문입니다. 프롬프트는 질문, 명령, 문장 시작 부분 등 다양한 형태를 취할 수 있으며, LLM의 응답을 유도하는 데 중요한 역할을 합니다.
- LLM(Large Language Model): GPT, Gemini 등 대규모 언어 모델로, 대량의 텍스트 데이터에서 학습하여 언어를 이해하고 생성할 수 있는 인공지능 시스템입니다. LLM은 프롬프트를 바탕으로 적절한 응답을 생성하거나, 주어진 작업을 수행하는 데 사용됩니다.

2. Basic LLM Chain

2) 일반적인 작동 방식

- **프롬프트 생성:** 사용자의 요구 사항이나 특정 작업을 정의하는 프롬프트를 생성합니다. 이 프롬프트는 LLM에게 전달되기 전에, 작업의 목적과 맥락을 명확히 전달하기 위해 최적화될 수 있습니다.

- **LLM 처리:** LLM은 제공된 프롬프트를 분석하고, 학습된 지식을 바탕으로 적절한 응답을 생성합니다. 이 과정에서 LLM은 내부적으로 다양한 언어 패턴과 내외부 지식을 활용하여, 요청된 작업을 수행하거나 정보를 제공합니다.

- **응답 반환:** LLM에 의해 생성된 응답은 최종 사용자에게 필요한 형태로 변환되어 제공됩니다. 이 응답은 직접적인 답변, 생성된 텍스트, 요약된 정보 등 다양한 형태를 취할 수 있습니다.

2. Basic LLM Chain

3) 실습 예제

- OpenAI의 ChatOpenAI 함수를 사용하면 GPT-3.5, GPT-4 모델을 API로 접속할 수 있습니다. 다음 예제는 랭체인에서 **GPT-3.5 모델을 사용하여 LLM 모델 인스턴스를 생성하고, "지구의 자전 주기는?" 라는 프롬프트를 LLM 모델에 전달하는 과정**을 보여줍니다. 실행 결과로 "지구의 자전 주기는?"에 대한 답변을 반환합니다. 이는 GPT 모델이 자연어 처리를 통해 질문의 의도를 이해하고, 학습된 데이터를 바탕으로 적절한 응답을 생성하기 때문입니다.

```
from langchain_openai import ChatOpenAI

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

# chain 실행
llm.invoke("지구의 자전 주기는?")
```

```
AIMessage(content='지구의 자전 주기는 약 23시간 56분 4초입니다. 이것은 지구가 자전하는 데 걸리는 시간을 의미합니다.
이 자전 주기는 하루의 길이를 결정하며, 이것이 하루가 24시간이 되도록 조정하기 위해 세계시가 도입되었습니다.')
```

3. Multi-Chain

멀티 체인 (Multi-Chain)

여러 개의 체인을 연결하거나 복합적으로 작용하는 것은 멀티 체인(Multi-Chain) 구조를 통해 이루어집니다.이러한 구조는 각기 다른 목적을 가진 여러 체인을 조합하여, 입력 데이터를 다양한 방식으로 처리하고 최종적인 결과를 도출할 수 있도록 합니다. 복잡한 데이터 처리, 의사 결정, AI 기반 작업 흐름을 설계할 때 특히 유용합니다.

3. Multi-Chain

멀티 체인 (Multi-Chain)

1. 순차적인 체인 연결

다음 예제를 통해서 2개의 체인(chain1, chain2)를 정의하고, 순차적으로 체인을 연결하여 수행하는 작업을 보겠습니다. 여기서 첫 번째 체인(chain1)은 한국어 단어를 영어로 번역하는 작업을 수행합니다.

```
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"})
```

```
future
```

3. Multi-Chain

멀티 체인 (Multi-Chain)

chain1에서 출력한 값("미래"라는 단어의 영어 번역)을 입력값으로 받아서, 이 번역된 단어를 english_word 변수에 저장합니다.

다음으로, 이 변수를 사용하여 두 번째 체인(chain2)의 입력으로 제공하고, 영어 단어의 뜻을 한국어로 설명하는 작업을 수행합니다. 최종 출력은 StrOutputParser()를 통해 모델의 출력을 문자열로 표시됩니다.

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

chain2.invoke({"korean_word":"미래"})
```

3. Multi-Chain

멀티 체인 (Multi-Chain)

미래란 무엇입니까?

미래란 앞으로 오는 시간이나 날이다. 미래는 아직 일어나지 않은 일들이나 상황을 나타낸다. 예를 들어, 내일이나 다음 주에 일어날 일들은 미래에 속한다. 미래는 예측하거나 예상할 수 있는 것이 아니기 때문에 불확실하고 불투명한 측면이 있다.하지만 우리는 미래를 준비하고 계획하기 위해 노력하고 예측하려고 노력한다.미래는 우리의 선택과 행동에 따라 변화할 수 있으며, 우리의 노력과 노력에 따라 좋은 결과를 가져올 수도 있다.미래는 우리가 살아가는 현재와 연결되어 있으며, 우리의 행동과 선택이 미래를 결정하는 요소가 될 수 있다.

Part 2.

Introduction to RAG

1. Definition

RAG(검색 증강 생성)이란?

- R에 해당하는 Retrieval:
  - 사용자의 질문이나 컨텍스트를 입력으로 받아서, **이와 관련된 외부 데이터를 검색**하는 단계
  - 이 때 검색 엔진이나 데이터베이스 등 다양한 소스에서 필요한 정보를 찾아냅니다. 검색된 데이터는 질문에 대한 답변을 생성하는데 적합하고 상세한 정보를 포함하는 것을 목표로 합니다.

- A에 해당하는 Augmented:
  - "증강되었다". 즉 생성 모델에 추가적인 정보(검색된 정보)를 제공하여 그 성능을 향상시키는 개념을 말합니다.

- G 에 해당하는 Generation:
  - 검색된 데이터를 기반으로 LLM 모델이 **사용자의 질문에 답변을 생성**하는 단계
  - 단계에서 모델은 검색된 정보와 기존의 지식을 결합하여, 주어진 질문에 대한 답변을 생성

1. Definition

RAG(검색 증강 생성)이란?

- RAG(Retrieval-Augmented Generation) 기법은 기존의 대규모 언어 모델(LLM)을 확장하여, 주어진 컨텍스트나 질문에 대해 더욱 정확하고 풍부한 정보를 제공하는 방법입니다.
- **모델이 학습 데이터에 포함되지 않은 외부 데이터를 실시간으로 검색(retrieval)하고, 이를 바탕으로 답변을 생성(generation)하는 과정을** 포함합니다.
- 특히 **환각**(생성된 내용이 사실이 아닌 것으로 오인되는 현상)을 **방지**하고, 모델이 최신 정보를 반영하거나 더 넓은 지식을 활용할 수 있게 합니다.
- LLM 단점을 보완하는 기술

1. Definition

RAG 모델의 장점

- **풍부한 정보 제공**: RAG 모델은 검색을 통해 얻은 외부 데이터를 활용하여, 보다 구체적이고 풍부한 정보를 제공할 수 있습니다.
- **실시간 정보 반영**: 최신 데이터를 검색하여 반영함으로써, 모델이 실시간으로 변화하는 정보에 대응할 수 있습니다.
- **환각 방지**: 검색을 통해 실제 데이터에 기반한 답변을 생성함으로써, 환각 현상이 발생할 위험을 줄이고 정확도를 높일 수 있습니다.

2. Pipeline

RAG의 pipeline

RAG(Retrieval-Augmented Generation) 파이프라인은 크게 아래 단계로 구성됩니다.

① 데이터 로드(Load Data)

② 텍스트 분할(Text Split)

③ 임베딩(Embedding)

④ 벡터저장(Vector Store)

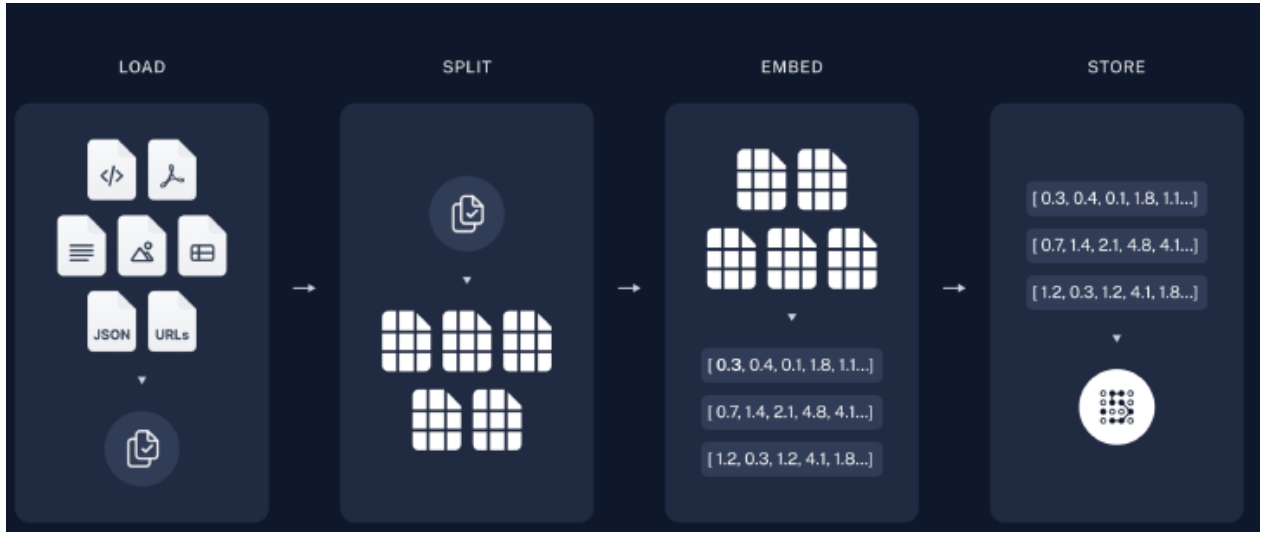
⑤ 검색(Retrieval)

⑥ 생성(Generation)

2. Pipeline

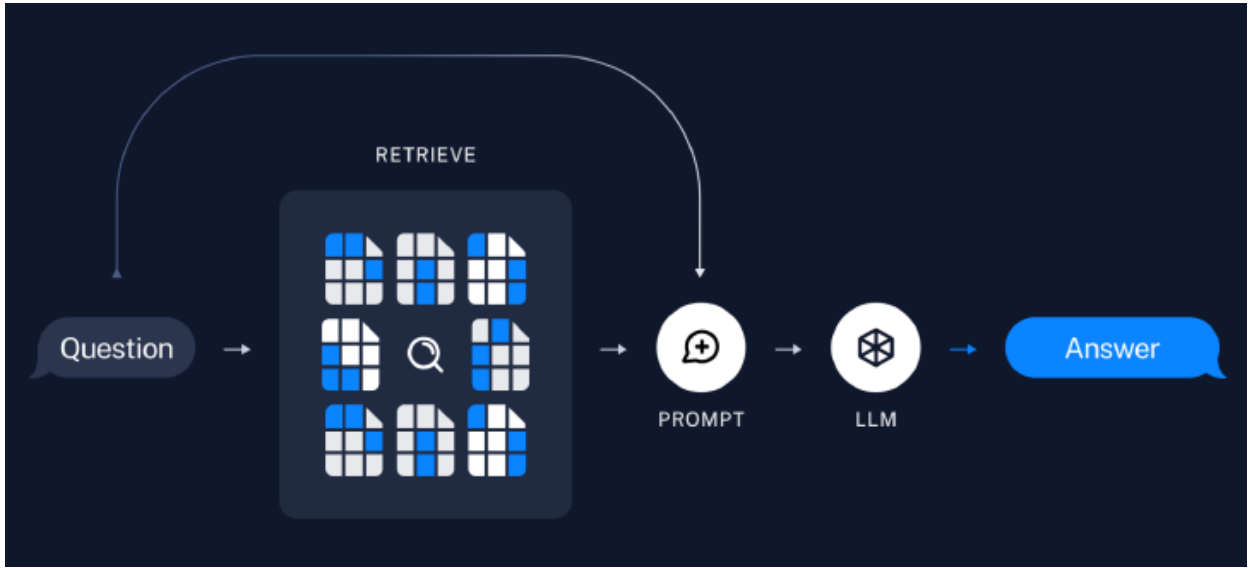
전처리 작업

① 데이터 로드(Load Data)
② 텍스트 분할(Text Split)
③ 임베딩(Embedding)
④ 벡터 저장(Vector Store)



서비스 단계에서 이뤄지는 작업

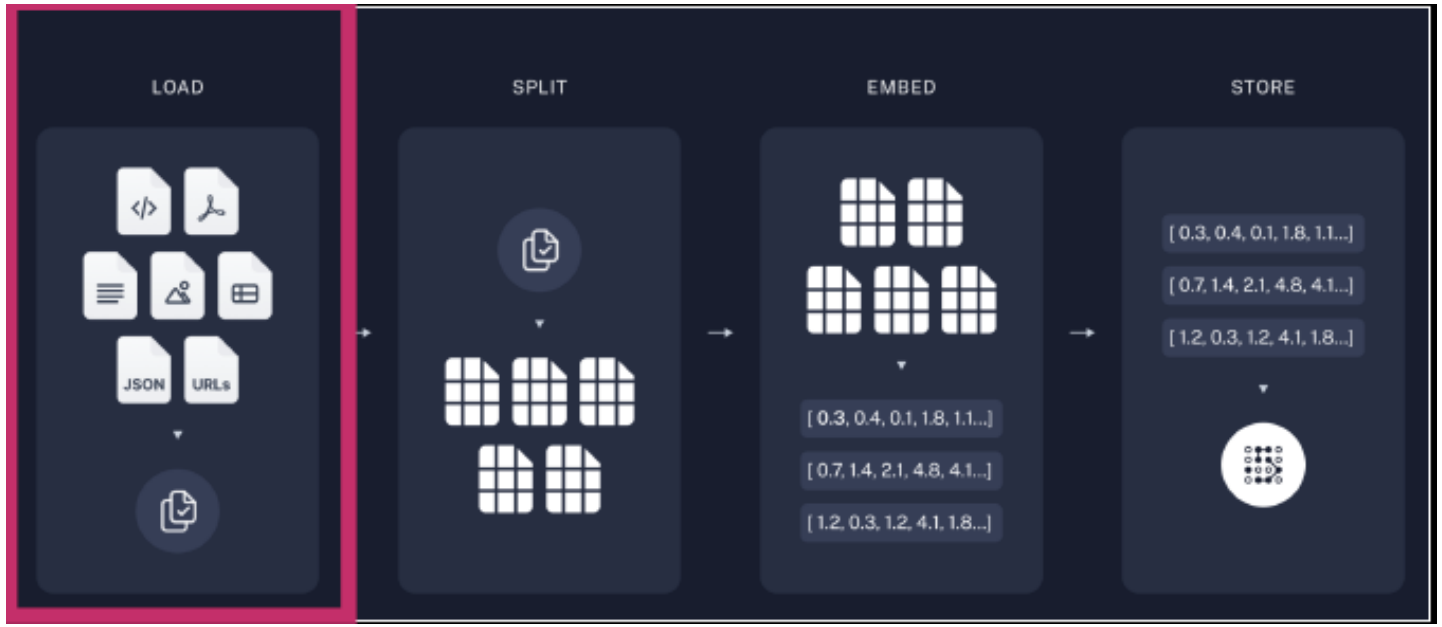
⑤ 검색(Retrieval)
⑥ 생성(Generation)



2. Pipeline

① 데이터 로드(Load Data)

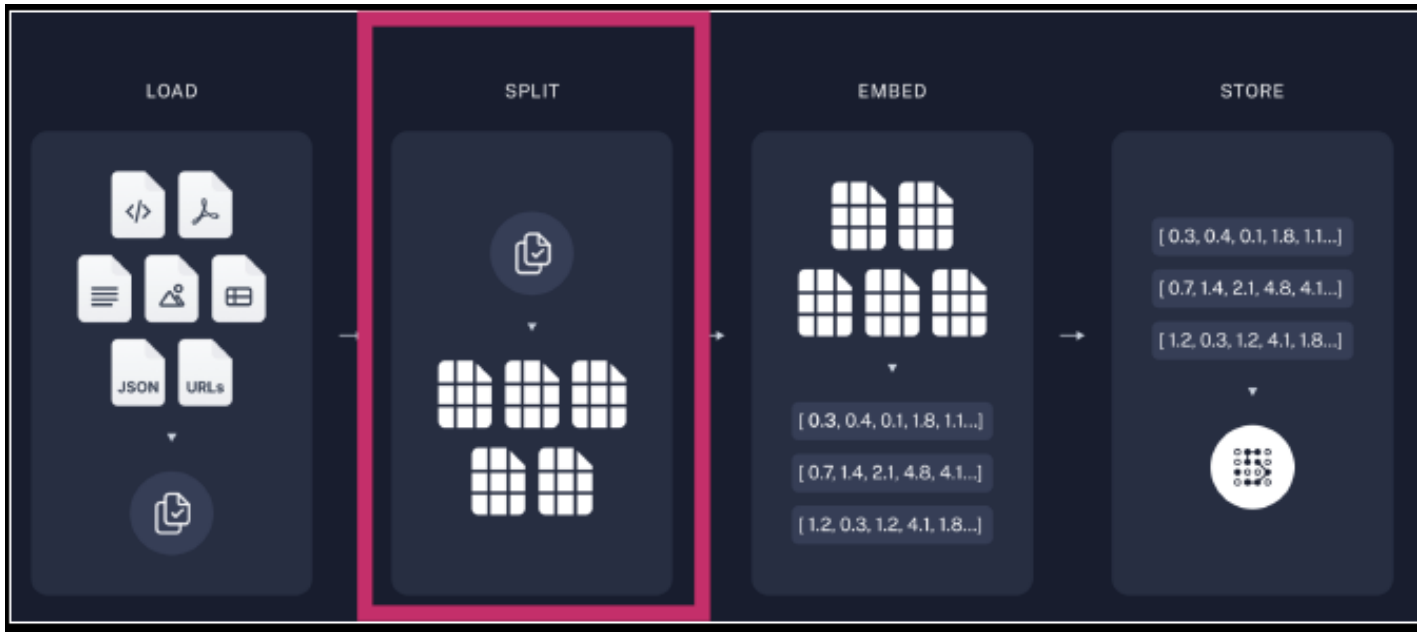
RAG에 **사용할 데이터를 불러오는 단계**입니다. 외부 데이터 소스에서 정보를 수집하고, 필요한 형식으로 변환하여 시스템에 로드합니다. 예를 들면 공개 데이터셋, 웹 크롤링을 통해 얻은 데이터, 또는 사전에 정리된 자료일 수 있습니다. 가져온 데이터는 검색에 사용될 지식이나 정보를 담고 있어야 합니다.



2. Pipeline

② 텍스트 분할(Text Split)

불러온 **데이터를 작은 크기의 단위(chunk)로 분할**하는 과정입니다. 자연어 처리(NLP) 기술을 활용하여 큰 문서를 처리가 쉽도록 문단, 문장 또는 구 단위로 나누는 작업입니다. **검색 효율성을 높이기 위한 중요한 과정입니다.**



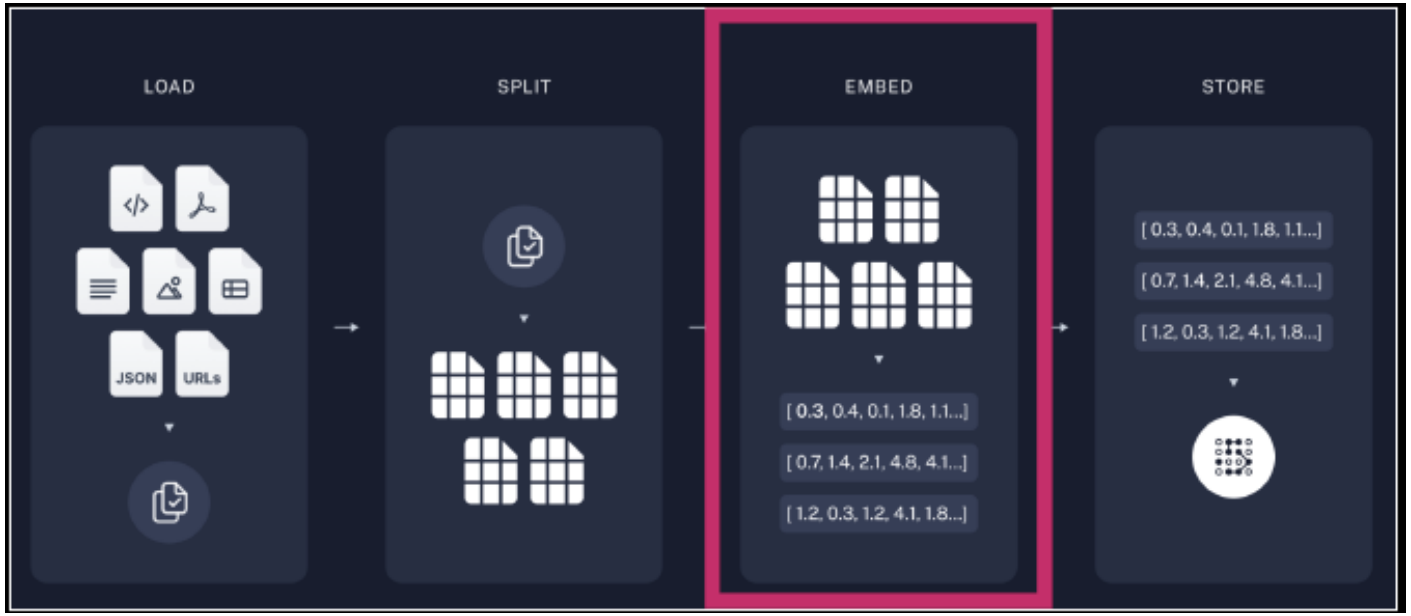
2. Pipeline

③ 임베딩(Embedding)

임베딩이란 **자연어**를 기계가 이해할 수 있는 숫자의 나열인 **벡터로 변환**하는 과정

분할된 텍스트를 검색 가능한 형태로 만드는 단계입니다.

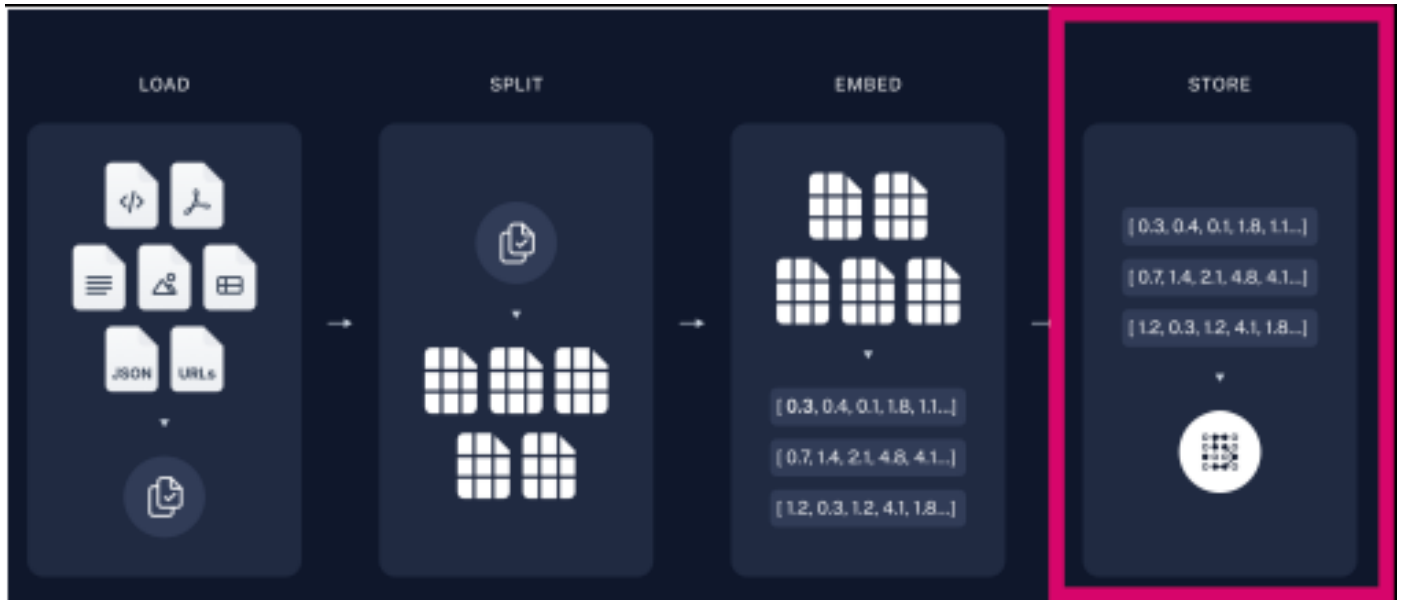
LangChain 라이브러리를 사용하여 텍스트를 벡터로 변환



2. Pipeline

④ 벡터 저장(Vector Store)

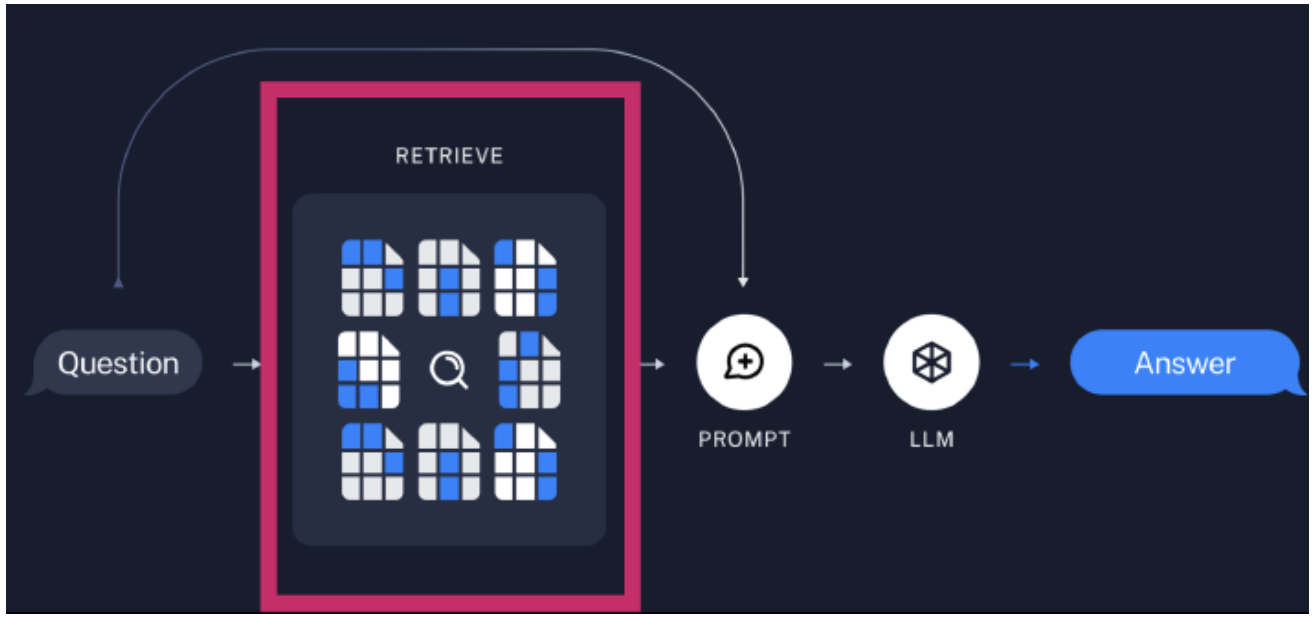
벡터를 Vector DB에 저장 하는 단계입니다. 앞 단계에서 변환된 벡터를 저장한 후, 저장된 임베딩을 기반으로 유사성 검색을 수행하는 과정을 진행합니다. 벡터 데이터베이스는 방대한 데이터 유입을 효율적으로 관리하는데 중요한 역할을 합니다.



2. Pipeline

⑤ 검색(Retrieval)

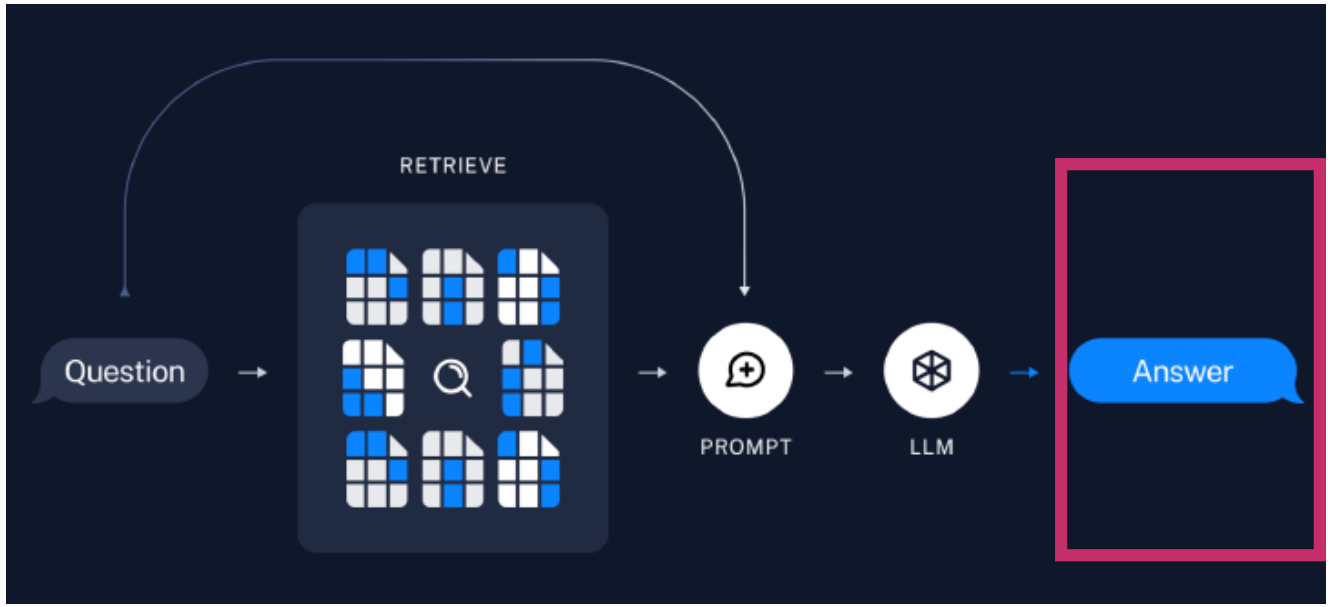
사용자의 질문이나 주어진 컨텍스트에 가장 관련된 정보를 찾아내는 과정입니다. 사용자의 입력을 바탕으로 쿼리를 생성하고, 데이터에서 **가장 관련성 높은 정보를 검색**합니다. LangChain의 retriever 메소드를 사용합니다.



2. Pipeline

⑥ 생성(Generation)

검색된 정보를 바탕으로 **사용자의 질문에 답변을 생성**하는 최종 단계입니다. LLM 모델에 검색 결과와 함께 사용자의 입력을 전달합니다. 모델은 사전 학습된 지식과 검색 결과를 결합하여 **주어진 질문에 가장 적절한 답변을 생성**합니다.



Part 3.

RAG with LangChain

0. RAG with LangChain

문서 전처리 단계

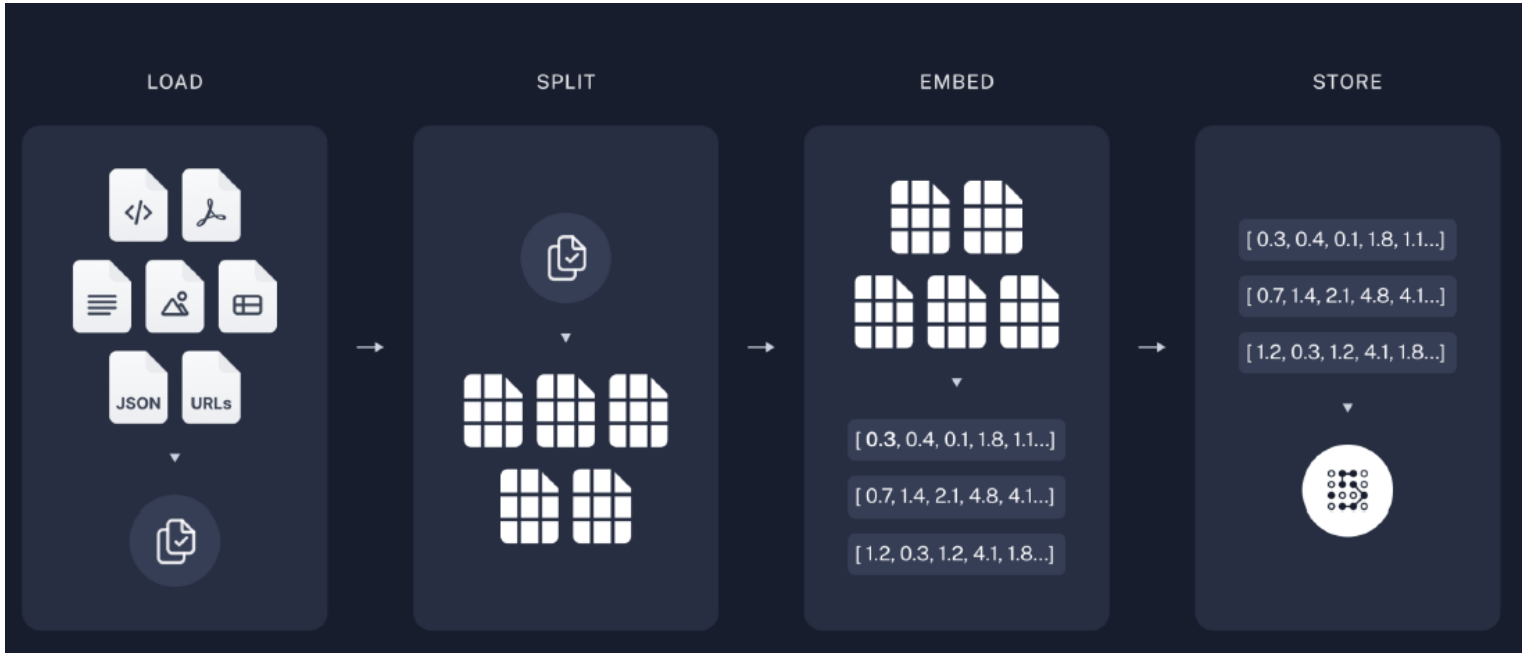
① Document Loader: 문서로드

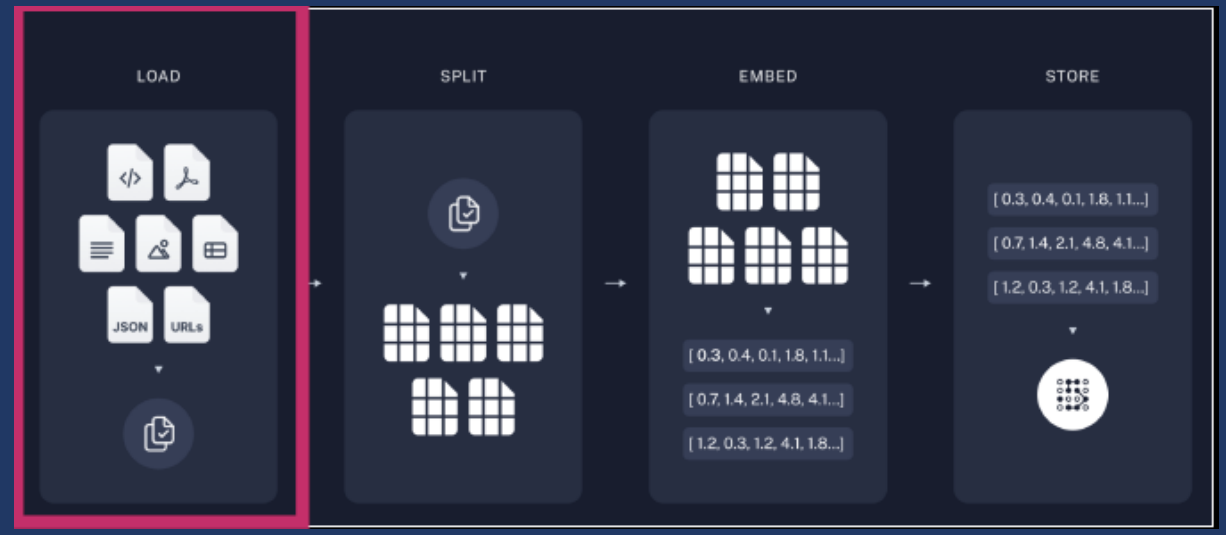
② Text Splitter: 분할 전략

③ Embedding: 임베딩

④ Vector Store: Vector DB

⑤ Retrievers: Vector DB 검색기





Document Loader

1. Document Loader

LangChain에서 Document Loader는 다양한 소스에서 **문서를 불러오고 처리하는 과정을 담당**합니다.

① **다양한 소스 지원**: 웹 페이지, PDF 파일, 데이터베이스 등 다양한 소스에서 문서를 불러올 수 있습니다.

② **데이터 변환 및 정제**: 랭체인의 다른 모듈이나 알고리즘이 처리하기 쉬운 형태로 변환합니다. 불필요한 데이터를 제거하거나, 구조를 변경할 수도 있습니다.

③ **효율적인 데이터 관리**: 대량의 문서 데이터를 효율적으로 관리하고, 필요할 때 쉽게 접근할 수 있도록 합니다. 이를 통해 검색 속도를 향상시키고, 전체 시스템의 성능을 높일 수 있습니다.

1. Document Loader

웹 문서 (WebBaseLoader)

웹 페이지에서 문서를 로드하기 위하여 WebBaseLoader를 사용

WebBaseLoader는 특정 웹 페이지의 내용을 로드하고 파싱하기 위해 설계된 클래스

```
import bs4
from langchain_community.document_loaders import WebBaseLoader

# 여러 개의 url 지정 가능
url1 = "https://blog.langchain.dev/week-of-1-22-24-langchain-release-notes/"
url2 = "https://blog.langchain.dev/week-of-2-5-24-langchain-release-notes/"

loader = WebBaseLoader(
    web_paths=(url1, url2),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("article-header", "article-content")
        )
    ),
)
docs = loader.load()
len(docs)
```

```
2
```

1. Document Loader

웹 문서 (WebBaseLoader)

첫 번째 문서의 내용을 출력하여 확인합니다.

```
docs[0]
```

```
Document(page_content='\n\uf8ffüóÉÔ∏è\xa0Introducing LangGraphWe just launched LangGraph, which helps customi
ze your Agent Runtime. You can read more about it on the blog.LangGraph helps construct a powerful agent exec
utor that allows for loops in logic while keeping track of application state. Agents constructed with LangGra
ph can handle more ambiguous inputs and accomplish tasks more consistently. These agents can:Force call a too
lWait for human-in-the-loop approvalManage agent stepsReturn outputs in a specific or raw formatWe,Äôre excit
ed to give our agents more controllable power. Learn about LangGraph by reading our docs. If you have feedbac
k, we,Äôd love to know!\uf8ffü¶ú\uf8ffüîó\xa0LangChain 0.1 is here!If you haven,Äôt yet upgraded to LangChain
0.1, you should check it out. We wrote more about it on our blog, but here are the big takeaways:No more brea
king changes on a minor version release.LCEL supports parallelization, fallbacks, batch, streaming, and async
all out-of-the-box.LangChain supports almost 700 integrations now and has improved co-maintenance with integr
ation partners.More advanced retrieval to get your RAG apps behaving better.Largest library of tools and temp
lates of cognitive architectures for your LangChain Agents.1-click observability with LangSmith.New in OSSInt
roducing the stream_events methodThis new method streams the current step the agent is taking, retrieved sour
ces at the step, and the agent,Äôs response. See docs.New and improved use case docsQ&A with RAG: Get started
with your own question answer chat bot.Text to SQL: We hear of lots of companies building their own text to S
QL bots; this guide will get you started.Tool use: Let your LLMs have access to APIs and functions to take ac
tion and integrate with existing company software.\uf8ffü¶ú\uf8ffüõ†Ô∏è\xa0New in LangSmithTracing improvemen
tsWe refreshed the run tree so that the most important calls are displayed in the default view. Keeping trace
s clean and uncluttered is an important step to getting your bearing on what,Äôs happening faster. If you wan
t more detail, you can always show all in an expanded view.Better rendering for tools and function calls in t
he trace view and playground.Use our keyboard shortcuts (J, K) to navigate to the previous or next run while
in the trace history view.Trace histories now have infinite scroll, so you can get more context.Annotation im
provementsFaster updates and deletes on annotation labels.Monitoring improvementsDrill downs and dynamic stat
calculations by tag on the monitoring tab, so that you filter charts by a segment.Notice faster rendering of
monitoring charts? We,Äôve re-platformed our database so we can keep scaling and maintain a snappy experience
in the UI.\uf8ffüëÄ\xa0Stay in the KnowEducational Resources:Build LLM Apps with LangChain.js by DeepLearnin
g.AILangGraph YouTube SeriesLangChain 0.1 YouTube SeriesStreaming Events with stream_events methodChat your P
DFs by Austin VanceGood Reads:Multi-agent WorkflowsAlly Financial Collaborates with LangChain to bring PII Ma
sking into LangChain JSBuilding a Wikipedia Chatbot Using AstraDB, LangChain, and Vercel by Carter Rabasa at
Datastax10 Lessons from Developing an AI Chatbot Using Retrieval-Augmented Generation by FiddlerHow Mendable
leverages LangSmith to debug Tools & Actions by Nicolas Camara at MendableEvents you should attend:Jan 30th:
Build LLM Data Apps with LangChain and DashFeb 8th: Streamlit,Äôs SF Meetup\n', metadata={'source': 'https://
blog.langchain.dev/week-of-1-22-24-langchain-release-notes/'})
```

1. Document Loader

텍스트 문서 (TextLoader)

TextLoader 이용하여 텍스트 파일 데이터 가져오기

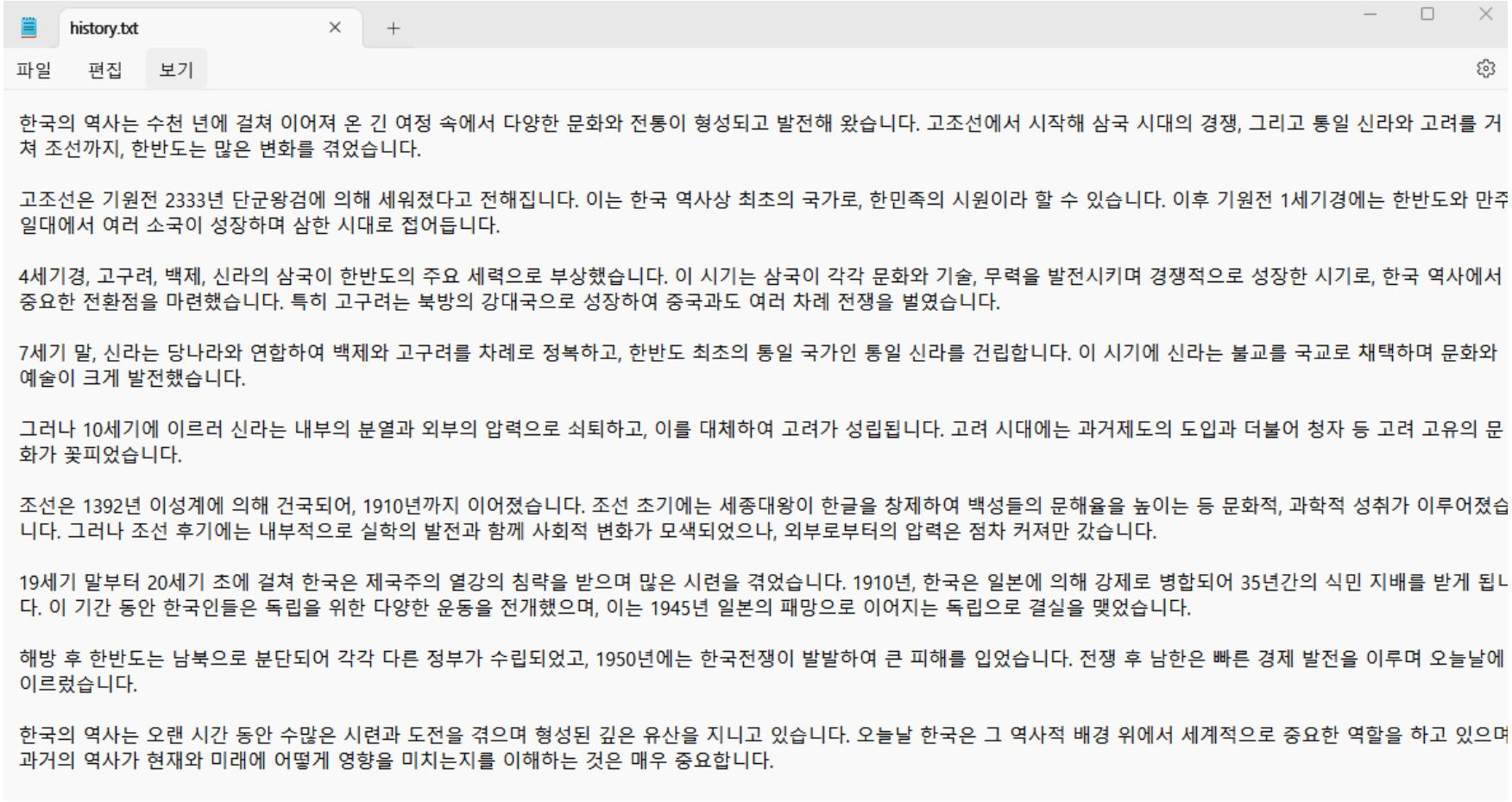
그리고 텍스트 파일의 내용을 랭체인의 Document 객체로 변환하고 이를 리스트 형태로 반환합니다.

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader('history.txt')
data = loader.load()

print(type(data))
print(len(data))
```

```
<class 'list'>
1
```

1. Document Loader

텍스트 문서 (TextLoader)



1. Document Loader

텍스트 문서 (TextLoader)

다음 출력 결과를 보면 리스트 배열 안에 Document 객체가 담겨 있는 것을 볼 수 있습니다.

Document 객체의 metadata 속성에는 원본 파일에 대한 메타데이터가 들어 있습니다. 원본 파일의 경로를 확인할 수 있습니다.

```
data
```

```
[Document(page_content='한국의 역사는 수천 년에 걸쳐 이어져 온 긴 여정 속에서 다양한 문화와 전통이 형성되고 발전해 왔습니다. 고조선에서 시작해 삼국 시대의 경쟁, 그리고 통일 신라와 고려를 거쳐 조선까지, 한반도는 많은 변화를 겪었습니다.\n\n고조선은 기원전 2333년 단군왕검에 의해 세워졌다고 전해집니다. 이는 한국 역사상 최초의 국가로, 한민족의 시원이라 할 수 있습니다. 이후 기원전 1세기경에는 한반도와 만주 일대에서 여러 소국이 성장하며 삼한 시대로 접어듭니다.\n\n4세기경, 고구려, 백제, 신라의 삼국이 한반도의 주요 세력으로 부상했습니다. 이 시기는 삼국이 각각 문화와 기술, 무력을 발전시키며 경쟁적으로 성장한 시기로, 한국 역사에서 중요한 전환점을 마련했습니다. 특히 고구려는 북방의 강대국으로 성장하여 중국과도 여러 차례 전쟁을 벌였습니다.\n\n7세기 말, 신라는 당나라와 연합하여 백제와 고구려를 차례로 정복하고, 한반도 최초의 통일 국가인 통일 신라를 건립합니다. 이 시기에 신라는 불교를 국교로 채택하며 문화와 예술이 크게 발전했습니다.\n\n그러나 10세기에 이르러 신라는 내부의 분열과 외부의 압력으로 쇠퇴하고, 이를 대체하여 고려가 성립됩니다. 고려 시대에는 과거제도의 도입과 더불어 청자 등 고려 고유의 문화가 꽃피었습니다.\n\n조선은 1392년 이성계에 의해 건국되어, 1910년까지 이어졌습니다. 조선 초기에는 세종대왕이 한글을 창제하여 백성들의 문해율을 높이는 등 문화적, 과학적 성취가 이루어졌습니다. 그러나 조선 후기에는 내부적으로 실학의 발전과 함께 사회적 변화가 모색되었으나, 외부로부터의 압력은 점차 커져만 갔습니다.\n\n19세기 말부터 20세기 초에 걸쳐 한국은 제국주의 열강의 침략을 받으며 많은 시련을 겪었습니다. 1910년, 한국은 일본에 의해 강제로 병합되어 35년간의 식민 지배를 받게 됩니다. 이 기간 동안 한국인들은 독립을 위한 다양한 운동을 전개했으며, 이는 1945년 일본의 패망으로 이어지는 독립으로 결실을 맺었습니다.\n\n해방 후 한반도는 남북으로 분단되어 각각 다른 정부가 수립되었고, 1950년에는 한국전쟁이 발발하여 큰 피해를 입었습니다. 전쟁 후 남한은 빠른 경제 발전을 이루며 오늘날에 이르렀습니다.\n\n한국의 역사는 오랜 시간 동안 수많은 시련과 도전을 겪으며 형성된 깊은 유산을 지니고 있습니다. 오늘날 한국은 그 역사적 배경 위에서 세계적으로 중요한 역할을 하고 있으며, 과거의 역사가 현재와 미래에 어떻게 영향을 미치는지를 이해하는 것은 매우 중요합니다.', metadata={'source': './history.txt'})]
```

```
len(data[0].page_content)
```

```
1234
```

```
data[0].metadata
```

```
{'source': 'history.txt'}
```

1. Document Loader

기타

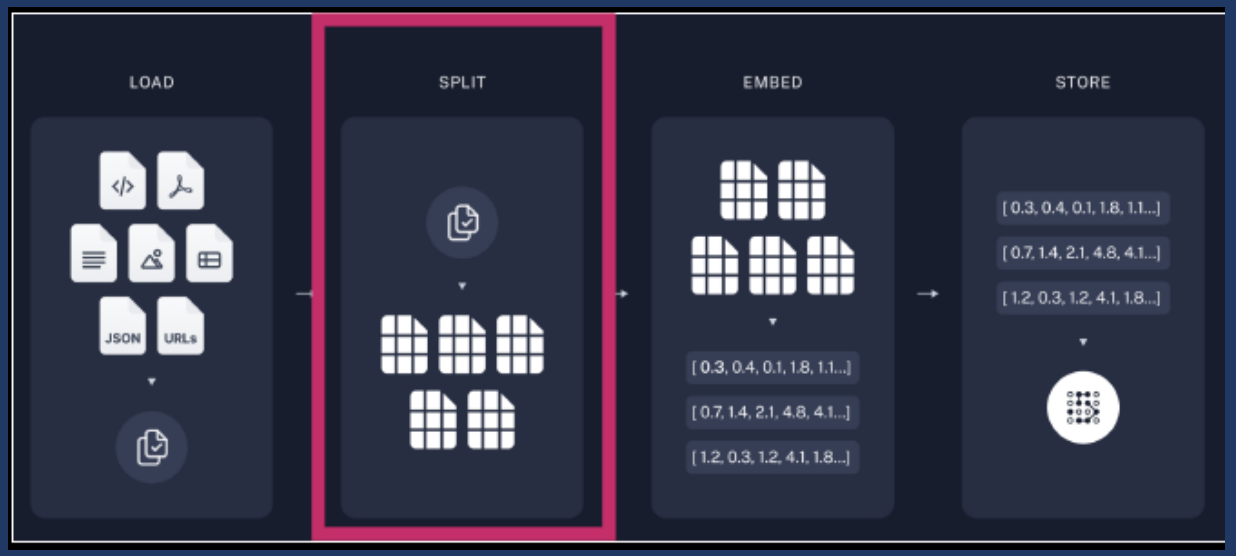
디렉토리 폴더 (DirectoryLoader): 특정 폴더의 모든 파일을 가져오기

CSV 문서 (CSVLoader): CSV 파일 데이터 가져오기

PDF 문서 (PDFLoader): PDF 파일 데이터 가져오기

등 다양한 로더가 존재

Text Splitter



2. Text Splitter

LangChain은 **긴 문서를 작은 단위인 청크(chunk)로 나누는 텍스트 분리 도구**를 다양하게 지원합니다. 텍스트를 분리하는 작업을 **청킹(chunking)**이라고 부르기도 합니다. **이렇게 문서를 작은 조각으로 나누는 이유는 LLM 모델의 입력 토큰의 개수가 정해져 있기 때문**입니다.

텍스트가 너무 긴 경우에는 핵심 정보 이외에 불필요한 정보들이 많이 포함될 수 있어서 RAG 품질이 낮아지는 요인이 될 수도 있습니다. 핵심 정보가 유지될 수 있는 적절한 크기로 나누는 것이 매우 중요합니다.

2. Text Splitter

텍스트 분리기(Text Splitter)는 분할하려는 텍스트 유형과 사용 사례에 맞춰 선택할 수 있는 다양한 옵션이 제공됩니다. 크게 두 가지 차원에서 검토가 필요합니다.

1) 텍스트가 어떻게 분리되는지:

텍스트를 나눌 때 각 청크가 독립적으로 의미를 갖도록 나눠야 합니다. 이를 위해 문장, 구절, 단락 등 문서 구조를 기준으로 나눌 수 있습니다.

2) 청크 크기가 어떻게 측정되는지:

각 청크의 크기를 직접 조정할 수 있습니다. LLM 모델의 입력 크기와 비용 등을 종합적으로 고려하여 애플리케이션에 적합한 최적 크기를 결정하는 기준입니다. 예를 들면 단어 수, 문자 수 등을 기준으로 나눌 수 있습니다.

2. Text Splitter

CharacterTextSplitter

CharacterTextSplitter 클래스는 주어진 **텍스트를 문자 단위로 분할**하는 데 사용

파이썬의 split 함수라고 생각하시면 됩니다.

문서를 개별 문자를 단위로
나누기 (separator="")

```
# 각 문자를 구분하여 분할
from langchain_text_splitters import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = '',
    chunk_size = 500,
    chunk_overlap  = 100,
    length_function = len,
)

texts = text_splitter.split_text(data[0].page_content)

len(texts)
```

```
3
```

2. Text Splitter

CharacterTextSplitter

분할된 텍스트 조각 중에서 첫번째 청크의 길이를 확인해보면 정확하게 500자임을 알 수 있습니다.

```
len(texts[0])
```

```
500
```

첫 번째 청크의 내용을 출력해 보면, 500자가 되는 분할점에서 나눠지고 있습니다. 문장 중간에서 분할되기 때문에 맥락이 단절되는 문제가 있습니다. 이런 현상을 방지하기 위해서 적절한 크기의 chunk_overlap을 설정해 주는 것이 중요합니다.

```
texts[0]
```

```
한국의 역사는 수천 년에 걸쳐 이어져 온 긴 여정 속에서 다양한 문화와 전통이 형성되고 발전해 왔습니다. 고조선에서 시작해 삼국 시대의 경쟁, 그리고 통일 신라와 고려를 거쳐 조선까지, 한반도는 많은 변화를 겪었습니다.

고조선은 기원전 2333년 단군왕검에 의해 세워졌다고 전해집니다. 이는 한국 역사상 최초의 국가로, 한민족의 시원이라 할 수 있습니다. 이후 기원전 1세기경에는 한반도와 만주 일대에서 여러 소국이 성장하며 삼한 시대로 접어듭니다.

4세기경, 고구려, 백제, 신라의 삼국이 한반도의 주요 세력으로 부상했습니다. 이 시기는 삼국이 각각 문화와 기술, 무력을 발전시키며 경쟁적으로 성장한 시기로, 한국 역사에서 중요한 전환점을 마련했습니다. 특히 고구려는 북방의 강대국으로 성장하여 중국과도 여러 차례 전쟁을 벌였습니다.

7세기 말, 신라는 당나라와 연합하여 백제와 고구려를 차례로 정복하고, 한반도 최초의 통일 국가인 통일 신라를 건립합니다. 이 시기에 신라는 불교를 국교
```

2. Text Splitter

CharacterTextSplitter

문서를 줄바꿈을 기준으로 나누기 (separator="₩n")

```
# 줄바꿈 문자를 기준으로 분할

text_splitter = CharacterTextSplitter(
    separator = '\n',
    chunk_size = 500,
    chunk_overlap  = 100,
    length_function = len,
)

texts = text_splitter.split_text(data[0].page_content)

len(texts)
```

```
3
```

2. Text Splitter

CharacterTextSplitter

분할된 각 청크의 길이를 확인해보면 정확하게 500자 단위로 나누어지지 않았습니다. 이처럼 줄바꿈 문자를 기준으로 최대 500자를 맞출 수 있는 위치를 찾아서 분할하게 됩니다.

```
len(texts[0]), len(texts[1]), len(texts[2])
```

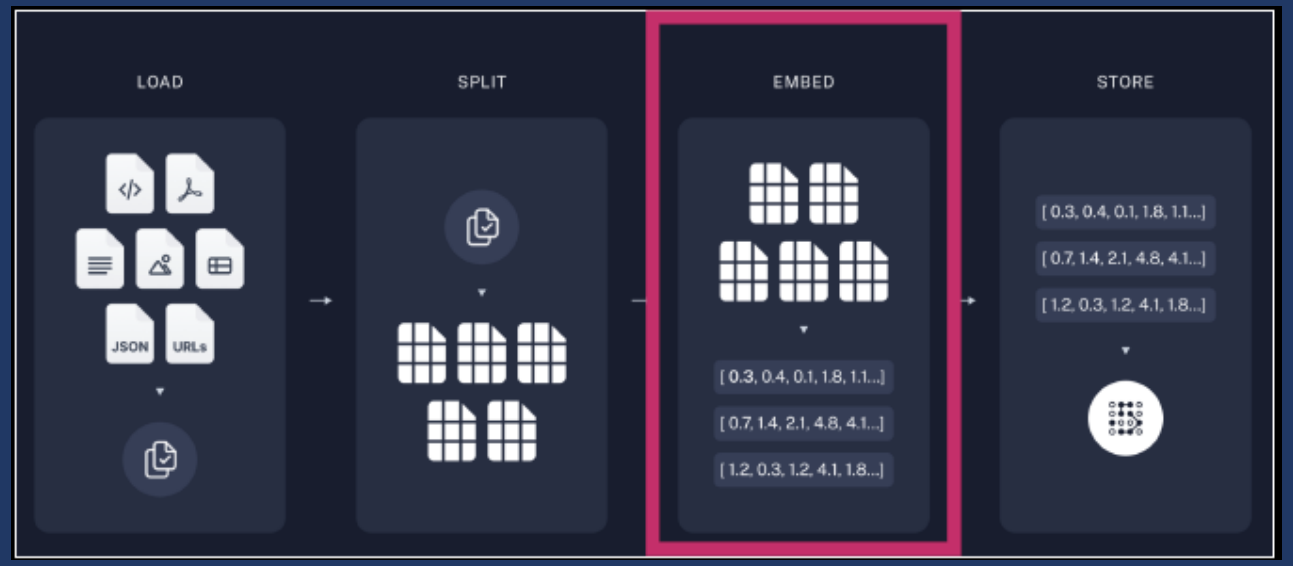
```
(411, 386, 427)
```

첫 번째 청크의 내용을 출력해 보면, 문장이 완전하게 끝나는 지점에서 분할되고 있습니다. 줄바꿈 문자가 있는 위치에서 나눠지기 때문입니다.

```
texts[0]
```

```
한국의 역사는 수천 년에 걸쳐 이어져 온 긴 여정 속에서 다양한 문화와 전통이 형성되고 발전해 왔습니다. 고조선에서 시작해 삼국 시대의 경쟁, 그리고 통일 신라와 고려를 거쳐 조선까지, 한반도는 많은 변화를 겪었습니다.
고조선은 기원전 2333년 단군왕검에 의해 세워졌다고 전해집니다. 이는 한국 역사상 최초의 국가로, 한민족의 시원이라 할 수 있습니다. 이후 기원전 1세기경에는 한반도와 만주 일대에서 여러 소국이 성장하며 삼한 시대로 접어듭니다.
4세기경, 고구려, 백제, 신라의 삼국이 한반도의 주요 세력으로 부상했습니다. 이 시기는 삼국이 각각 문화와 기술, 무력을 발전시키며 경쟁적으로 성장한 시기로, 한국 역사에서 중요한 전환점을 마련했습니다. 특히 고구려는 북방의 강대국으로 성장하여 중국과도 여러 차례 전쟁을 벌였습니다.
```

Embedding


LOAD
SPLIT
EMBED
STORE
JSON
URLs
[0.3, 0.4, 0.1, 1.8, 1.1...]
[0.7, 1.4, 2.1, 4.8, 4.1...]
[1.2, 0.3, 1.2, 4.1, 1.8...]
[0.3, 0.4, 0.1, 1.8, 1.1...]
[0.7, 1.4, 2.1, 4.8, 4.1...]
[1.2, 0.3, 1.2, 4.1, 1.8...]

3. Embedding

임베딩(Embedding)은 **텍스트 데이터**를 숫자로 이루어진 **벡터로 변환**하는 과정을 말합니다. 이러한 벡터 표현을 사용하면, 텍스트 데이터를 벡터 공간 내에서 수학적으로 다룰 수 있게 되며, 이를 통해 텍스트 간의 **유사성을 계산**하거나, 텍스트 데이터를 기반으로 하는 다양한 머신러닝 및 자연어 처리 작업을 수행할 수 있습니다.

임베딩 과정은 텍스트의 의미적인 정보를 보존하도록 설계되어 있어, **벡터 공간에서 가까이 위치한 텍스트 조각들은 의미적으로도 유사한 것으로 간주**됩니다.

3. Embedding

임베딩의 주요 활용 사례:

의미 검색(Semantic Search): 벡터 표현을 활용하여 의미적으로 유사한 텍스트를 검색하는 과정으로, 사용자가 입력한 쿼리에 대해 가장 관련성 높은 문서나 정보를 찾아내는 데 사용됩니다.

문서 분류(Document Classification): 임베딩된 텍스트 벡터를 사용하여 문서를 특정 카테고리나 주제에 할당하는 분류 작업에 사용됩니다.

텍스트 유사도 계산(Text Similarity Calculation): 두 텍스트 벡터 사이의 거리를 계산하여, 텍스트 간의 유사성 정도를 정량적으로 평가합니다.

3. Embedding

임베딩 모델 제공자:

OpenAI: GPT와 같은 언어 모델을 통해 텍스트의 임베딩 벡터를 생성할 수 있는 API를 제공합니다.

Hugging Face: Transformers 라이브러리를 통해 다양한 오픈소스 임베딩 모델을 제공합니다.

Google: Gemini, Gemma 등 언어 모델에 적용되는 임베딩 모델을 제공합니다.

3. Embedding

임베딩 메소드:

embed_documents: 이 메소드는 문서 객체의 집합을 입력으로 받아, 각 문서를 벡터 공간에 임베딩합니다. 주로 대량의 텍스트 데이터를 배치 단위로 처리할 때 사용됩니다.

embed_query: 이 메소드는 단일 텍스트 쿼리를 입력으로 받아, 쿼리를 벡터 공간에 임베딩합니다. 주로 사용자의 검색 쿼리를 임베딩하여, 문서 집합 내에서 해당 쿼리와 유사한 내용을 찾아내는 데 사용됩니다.

임베딩은 텍스트 데이터를 머신러닝 모델이 이해할 수 있는 형태로 변환하는 핵심 과정입니다. 다양한 자연어 처리 작업의 기반이 되는 중요한 작업입니다.

3. Embedding

OpenAIEmbeddings

OpenAIEmbeddings 클래스 소개

OpenAIEmbeddings 클래스는 OpenAI의 API를 활용하여, 각 문서를 대응하는 임베딩 벡터로 변환합니다. langchain_openai 라이브러리에서 OpenAIEmbeddings 클래스를 직접 임포트합니다.

```
from langchain_openai import OpenAIEmbeddings

embeddings_model = OpenAIEmbeddings()
```

3. Embedding

OpenAIEmbeddings

문서 임베딩

임베딩 모델을 통해 다섯 개의 문장을 벡터로 변환합니다.

여기서 embed_documents 메서드는 입력된 문장들을 각각 임베딩 벡터로 변환합니다. embeddings 변수에는 5개의 벡터가 들어 있습니다. 각 벡터는 1536개의 숫자(차원)를 가지고 있습니다.

```
embeddings = embeddings_model.embed_documents(
    [
        '안녕하세요!',
        '어! 오랜만이에요',
        '이름이 어떻게 되세요?',
        '날씨가 추워요',
        'Hello LLM!'
    ]
)
len(embeddings), len(embeddings[0])
```

```
(5, 1536)
```

이 코드에서 len(embeddings)는 문장 개수(5개)를 반환하고, len(embeddings[0])는 첫 번째 문장의 벡터 크기(1536)를 반환합니다.

3. Embedding

OpenAIEmbeddings

임베딩 벡터 일부 출력

첫 번째 문서의 변환된 임베딩 벡터의 일부를 살펴봅니다. 이 코드는 첫 번째 문장의 임베딩 벡터의 앞쪽 20개의 값을 출력합니다. 벡터는 숫자들의 나열로 구성되어 있으며, 각각의 숫자는 문장의 의미를 특정 차원에서 표현합니다.

```
print(embeddings[0][:20])
```

```
[-0.01043144313417178, -0.01360457736091551, -0.006555278189843457, -0.01867146007935269, -0.018278775374135456, 0.016682708305029652, -0.009228057976315544, 0.003917333126600066, -0.007429315561219352, 0.010057760203067753, 0.011761499335534133, -0.006707284932166889, -0.025372409689831416, -0.022534956708611074, -0.004883207283914662, -0.021724255324649517, 0.0252964063186697, -0.01764541643131464, 0.007948670378620149, -0.017898760380971 99]
```

3. Embedding

OpenAIEmbeddings

쿼리 임베딩

embed_query 메소드는 단일 쿼리 문자열을 받아 이를 벡터 공간에 임베딩합니다. 주로 검색 쿼리나 질문 같은 단일 텍스트를 임베딩할 때 유용하며, 생성된 임베딩을 사용해 유사한 문서나 답변을 찾을 수 있습니다.

embedded_query[:5]는 생성된 임베딩 벡터의 처음 5개 원소를 슬라이싱하여 반환합니다. 임베딩의 일부 특성을 살펴볼 수 있습니다.

```
embedded_query = embeddings_model.embed_query('첫인사를 하고 이름을 물어봤나요?')
embedded_query[:5]
```

```
[0.003583437611976953,
 -0.024275508331077004,
 0.010910765516523958,
 -0.041075822901896646,
 -0.0044800835751111038]
```

```
[
    '안녕하세요!',
    '어! 오랜만이에요',
    '이름이 어떻게 되세요?',
    '날씨가 추워요',
    'Hello LLM!'
]
```

3. Embedding

OpenAIEmbeddings

코사인 유사도 계산

코사인 유사도는 두 벡터 간의 유사성을 측정하는 방법입니다. 코사인 유사도 값이 1에 가까울수록 두 벡터는 매우 유사하다는 것을 의미합니다. 이 함수는 두 벡터 A와 B 사이의 코사인 유사도를 계산합니다.

```
# 코사인 유사도
import numpy as np
from numpy import dot
from numpy.linalg import norm

def cos_sim(A, B):
  return dot(A, B)/(norm(A)*norm(B))
```

3. Embedding

OpenAIEmbeddings

문서와 쿼리의 유사도 계산

앞에서 임베딩한 문서들과 쿼리 벡터 사이의 코사인 유사도를 계산합니다.

이 루프는 모든 문서의 임베딩 벡터와 쿼리 벡터 간의 코사인 유사도를 계산하고 그 결과를 출력합니다

```
for embedding in embeddings:
    print(cos_sim(embedding, embedded_query))
```

3. Embedding

OpenAIEmbeddings

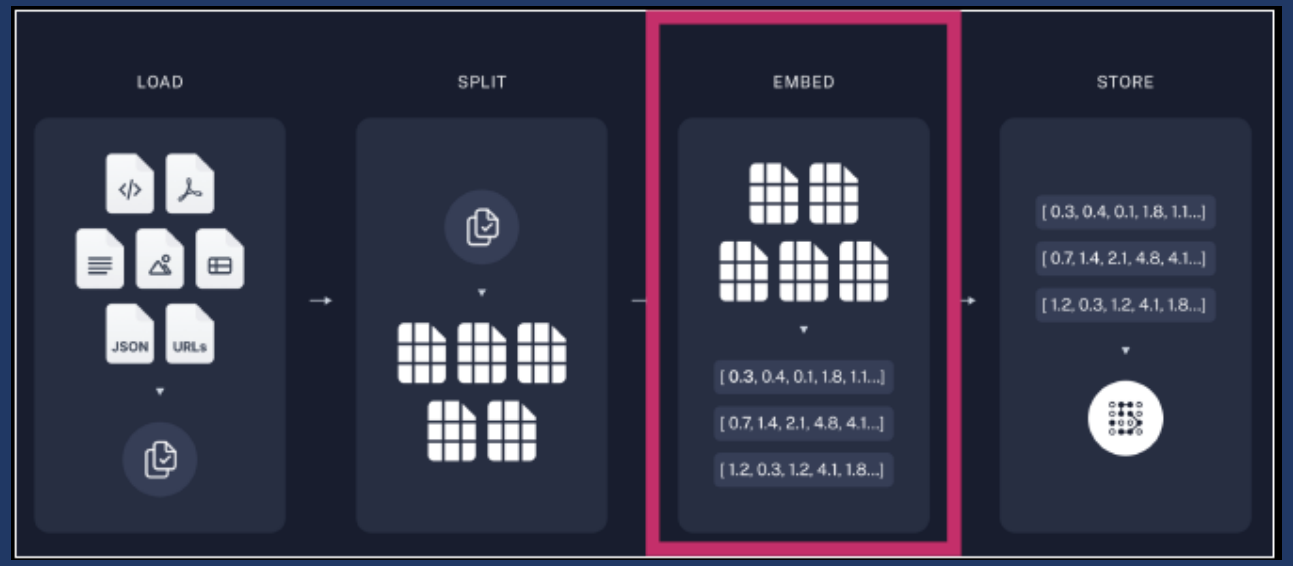
결과 해석

유사도 결과는 문서가 쿼리와 얼마나 관련이 있는지를 나타냅니다. 유사도가 높은 문서일수록 쿼리와 더 관련이 깊다고 볼 수 있습니다.

여기서 **세 번째 문서(유사도 0.8844988896647352)가 쿼리와 가장 유사**하며, 마지막 문서(유사도 0.7467444893657921)가 쿼리와 가장 덜 유사합니다.이러한 방식으로 문서 검색, 추천 시스템 등에서 유사도를 기준으로 문서를 필터링하거나 정렬할 수 있습니다.

```
0.8348269880307686
0.815446453577144
0.8844988896647352
0.7898616245127719
0.7467444893657921
```

Vector Store



4. Vector Store

벡터 저장소(Vector Store)는 벡터 형태로 표현된 데이터, 즉 **임베딩 벡터들을 효율적으로 저장하고 검색할 수 있는 시스템이나 데이터베이스**를 의미합니다. 자연어 처리(NLP), 이미지 처리, 그리고 기타 다양한 머신러닝 응용 분야에서 생성된 고차원 벡터 데이터를 관리하기 위해 설계되었습니다. 벡터 저장소의 핵심 기능은 대규모 벡터 데이터셋에서 빠른 속도로 가장 유사한 항목을 찾아내는 것입니다.

4. Vector Store

벡터 저장

임베딩 벡터는 텍스트, 이미지, 소리 등 다양한 형태의 데이터를 벡터 공간에 매핑한 것으로, 데이터의 의미적, 시각적, 오디오적 특성을 **수치적으로 표현**합니다. 이러한 벡터를 효율적으로 저장하기 위해서는 고차원 벡터를 처리할 수 있도록 최적화된 데이터 저장 구조가 필요합니다.

벡터 검색

저장된 벡터들 중에서 **사용자의 쿼리에 가장 유사한 벡터를 빠르게 찾아내는 과정**입니다. 이를 위해 코사인 유사도, 유클리드 거리, 맨해튼 거리 등 다양한 유사도 측정 방법을 사용할 수 있습니다. 코사인 유사도는 방향성을 기반으로 유사도를 측정하기 때문에 텍스트 임베딩 검색에 특히 자주 사용됩니다.

결과 반환

사용자의 쿼리에 대해 계산된 **유사도 점수를 기반으로 가장 유사한 항목들을 순서대로 사용자에게 반환**합니다. 이 과정에서는 유사도 점수뿐만 아니라, 검색 결과의 관련성, 다양성, 신뢰도 등 다른 요소들을 고려할 수도 있습니다.

4. Vector Store

Chroma

Chroma는 임베딩 벡터를 저장하기 위한 오픈소스 소프트웨어로, LLM(대규모 언어 모델) 앱 구축을 용이하게 하는 핵심 기능을 수행합니다. Chroma의 주요 특징은 다음과 같습니다:

임베딩 및 메타데이터 저장: 대규모의 임베딩 데이터와 이와 관련된 메타데이터를 효율적으로 저장할 수 있습니다.

문서 및 쿼리 임베딩: 텍스트 데이터를 벡터 공간에 매핑하여 임베딩을 생성할 수 있으며, 이를 통해 검색 작업이 가능합니다.

임베딩 검색: 사용자 쿼리에 기반하여 가장 관련성 높은 임베딩을 찾아내는 검색 기능을 제공합니다.

4. Vector Store

Chroma 벡터 저장소를 사용하여 임베딩된 텍스트 데이터를 저장하고 검색하는 방법을 설명합니다.

Chroma 방법1:유사도 기반 검색 (Similarity search)

텍스트 데이터 로드 > 텍스트 분할 > 임베딩 모델 초기화 > Chroma 벡터 저장소 생성

Chroma.from_texts 메소드를 사용하여 분할된 텍스트들을 임베딩하고, 이 임베딩을 Chroma 벡터 저장소에 저장

저장소는 collection_name으로 구분되며, 여기서는 'history'라는 이름을 사용합니다.

저장된 데이터는 ./db/chromadb 디렉토리에 저장됩니다.

collection_metadata에서 'hnsw:space': 'cosine'을 설정하여 유사도 계산에 코사인 유사도를 사용합니다.

```
embeddings_model = OpenAIEmbeddings()
db = Chroma.from_texts(
    texts,
    embeddings_model,
    collection_name = 'history',
    persist_directory = './db/chromadb',
    collection_metadata = {'hnsw:space': 'cosine'}, # l2 is the default
)

db
```

```
<langchain_community.vectorstores.chroma.Chroma at 0x7d46ca6afc70>
```

4. Vector Store

유사도 기반 검색 수행

query 변수에 검색 쿼리를 정의합니다.

db.similarity_search 메소드를 사용하여 저장된 데이터 중에서 쿼리와 가장 유사한 문서를 찾습니다.

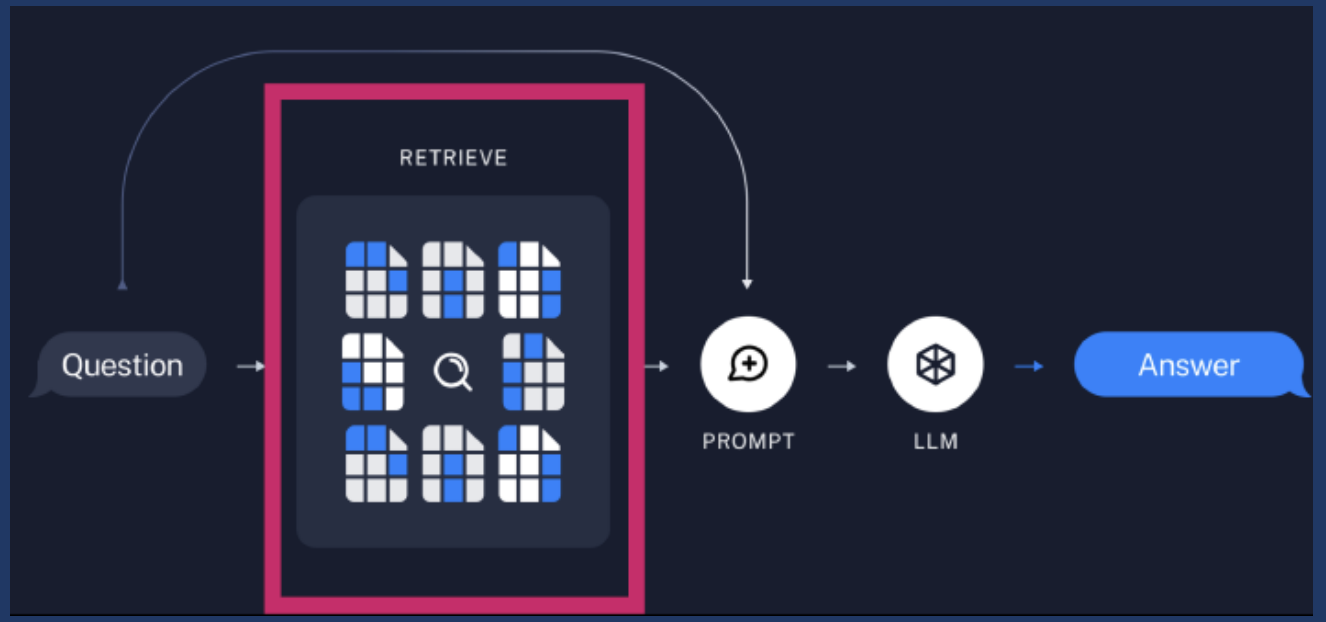
검색 결과를 docs 변수에 저장하고, 가장 유사한 문서의 내용은 docs[0].page_content를 통해 확인합니다.

이 과정을 통해, 주어진 쿼리('누가 한글을 창제했나요?')에 대해 가장 관련성 높은 텍스트 조각('...세종대왕이 한글을 창제하여...')을 찾아내고 있습니다.

```
query = '누가 한글을 창제했나요?'
docs = db.similarity_search(query)
print(docs[0].page_content)
```

```
조선은 1392년 이성계에 의해 건국되어, 1910년까지 이어졌습니다. 조선 초기에는 세종대왕이 한글을 창제하여 백성들의 문해율을 높이는 등 문화적, 과학적 성취가 이루어졌습니다. 그러나 조선 후기에는 내부적으로 실학의 발전과 함께 사회적 변화가 모색되었으나, 외부로부터의 압력은 점차 커져만 갔습니다.
```

Retriever



5. Retriever

Retrieval Augmented Generation (RAG)에서 **검색도구(Retrievers)** 는 벡터 저장소에서 문서를 검색하는 도구입니다. LangChain은 간단한 의미 검색도구부터 성능 향상을 위해 고려된 다양한 검색 알고리즘을 지원합니다.

LangChain에서 제공하는 검색도구(Retrievers) 에 대해서 알아보겠습니다.

5. Retriever

Vector Store Retriver

벡터스토어 검색도구(Vector Store Retriever)를 사용하면 대량의 텍스트 데이터에서 관련 정보를 효율적으로 검색할 수 있습니다.

다음 코드에서는 LangChain의 **벡터스토어**와 **임베딩 모델**을 사용하여 **문서들의 임베딩을 생성**하고,
그 후 저장된 임베딩들을 기반으로 **검색 쿼리에 가장 관련 있는 문서들을 검색**하는 방법을 설명합니다.

5. Retriever

Vector Store Retriver

사전 준비 - 문서 로드 및 분할 (document loader, text splitter)

```
# Load data -> Text split

from langchain_community.document_loaders import PyMuPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings

loader = PyMuPDFLoader('323410_카카오뱅크_2023.pdf')
data = loader.load()
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=1000,
    chunk_overlap=200,
    encoding_name='cl100k_base'
)

documents = text_splitter.split_documents(data)
len(documents)
```

- PyMuPDFLoader를 사용하여 PDF 파일('323410_카카오뱅크_2023.pdf')에서 텍스트 데이터를 로드합니다. 이 클래스는 PyMuPDF 라이브러리를 사용하여 PDF 문서의 내용을 추출합니다.
- RecursiveCharacterTextSplitter를 사용하여 문서를 텍스트 조각으로 분할하는 인스턴스를 생성하고 text_splitter.split_documents(data)를 호출하여 로드된 문서 객체를 여러 개의 청크로 분할합니다.
- documents 변수에는 모두 145개의 문서 조각으로 분할되어 저장됩니다.

```
145
```

5. Retriever

Vector Store Retriver

사전 준비 - 문서 임베딩을 벡터스토어에 저장 (embedding, vector store)

HuggingFaceEmbeddings를 사용하여 한국어 임베딩 모델인 'jhgan/ko-sbert-nli'를 사용합니다. 임베딩을 정규화하도록 설정합니다.

FAISS 벡터스토어를 사용하여 문서의 임베딩을 저장합니다. 여기서 코사인 유사도를 측정 기준으로 사용합니다.

```
# 벡터스토어에 문서 임베딩을 저장
from langchain_community.vectorstores import FAISS
from langchain_community.vectorstores.utils import DistanceStrategy
from langchain_community.embeddings import HuggingFaceEmbeddings

embeddings_model = HuggingFaceEmbeddings(
    model_name='jhgan/ko-sbert-nli',
    model_kwargs={'device':'cpu'},
    encode_kwargs={'normalize_embeddings':True},
)


vectorstore = FAISS.from_documents(documents,
                                   embedding = embeddings_model,
                                   distance_strategy = DistanceStrategy.COSINE
                                  )
```

5. Retriever

Vector Store Retriever

단일 문서 검색 (retriever)

검색 쿼리를 정의한 후, as_retriever 메소드를 사용하여 벡터스토어에서 Retriever 객체를 생성합니다.

search_kwargs에서 k: 1을 설정하여 가장 유사도가 높은 하나의 문서를 검색합니다.

```
# 검색 쿼리
query = '카카오뱅크의 환경목표와 세부추진내용을 알려줘'

# 가장 유사도가 높은 문장을 하나만 추출
retriever = vectorstore.as_retriever(search_kwargs={'k': 1})

docs = retriever.get_relevant_documents(query)
print(len(docs))
docs[0]
```

```
1
Document(page_content='더 나은 세상을 만들어 나가는데 앞장서겠습니다.\n이에 따라 카카오뱅크는 아래와 같은 환경방침을 수립하여 운영합니다.\n•\t전사 환경경영 정책 수립\n•\t녹색 구매 지침 수립\n•\t환경 지표 설정 및 성과 관리\n•\t용수, 폐기물, 에너지 등\n\t 자원 사용량 관리\n•\t기후변화를 포함한\n\t 환경 리스크 관리체계 마련\n•\tScope 1&2&3 온실가스\n\t 배출량 모니터링\n•\t탄소 가격 도입을 통한\n\t 환경 비용 관리\n•\t신재생 에너지 사용 확대\n•\t녹색채권 발행 기반 마련\n단기\n중기\n장기\n•\t환경경영 조직 및\n\t 관리 체계 구축\n•\t환경영향평가 체계 구축\n환경경영\n관리체계\n구축\n환경\n리스크 및\n성과 관리\n환경\n영향\n저감\n환경경영체계 구축\n카카오뱅크 환경방침\nProtect Environment\n카카오뱅크 2022 지속가능경영보고서', metadata={'source': '323410_카카오뱅크_2023.pdf', 'file_path': '323410_카카오뱅크_2023.pdf', 'page': 25, 'total_pages': 99, 'format': 'PDF 1.4', 'title': '', 'author': '', 'subject': '', 'keywords': '', 'creator': 'Adobe InDesign 18.1 (Macintosh)', 'producer': 'Adobe PDF Library 17.0', 'creationDate': "D:20230621170154+09'00'", 'modDate': "D:20230621170234+09'00'", 'trapped': ''})
```

5. Retriever

Vector Store Retriver

Generation - 답변 생성

이번에는 실제로 사용자 쿼리('카카오뱅크의 환경목표와 세부추진내용을 알려줘')에 대한 답변을 생성해보겠습니다. 벡터 저장소에서 문서를 검색한 다음, 이를 기반으로 ChatGPT 모델에 쿼리를 수행하는 end-to-end 프로세스를 구현합니다. 이 과정을 통해 사용자의 질문에 대한 의미적으로 관련이 있는 답변을 생성할 수 있습니다.

5. Retriever

Vector Store Retriver

Generation - 답변 생성

1)검색 (Retrieval)

2)프롬프트 생성 (Prompt)

3)모델 (Model)

4)문서 포맷팅 (Formatting Docs)

5)체인 실행 (Chain Execution)

6)실행 (Run)

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


# Retrieval
retriever = vectorstore.as_retriever(
    search_type='mmr',
    search_kwargs={'k': 5, 'lambda_mult': 0.15}
)

docs = retriever.get_relevant_documents(query)

# Prompt
template = '''Answer the question based only on the following context:
{context}

Question: {question}
'''

prompt = ChatPromptTemplate.from_template(template)
```

```
# Model
llm = ChatOpenAI(
    model='gpt-3.5-turbo-0125',
    temperature=0,
    max_tokens=500,
)


def format_docs(docs):
    return '\n\n'.join([d.page_content for d in docs])

# Chain
chain = prompt | llm | StrOutputParser()

# Run
response = chain.invoke({'context': (format_docs(docs)), 'question':query})
response
```

5. Retriever

Vector Store Retriver

Generation - 답변 생성

```
카카오뱅크의 환경목표는 '더 나은 세상을 만들어 나가는데 앞장서겠습니다'입니다. 이를 위해 카카오뱅크는 다음과 같은 환경 방침을 수립하여 운영하고 있습니다:\n- 전사 환경경영 정책 수립\n- 녹색 구매 지침 수립\n- 환경 지표 설정 및 성과 관리\n- 자원 사용량 관리 (용수, 폐기물, 에너지 등)\n- 기후변화를 포함한 환경 리스크 관리체계 마련\n- Scope 1&2&3 온실가스 배출량 모니터링\n- 탄소 가격 도입을 통한 환경 비용 관리\n- 신재생 에너지 사용 확대\n- 녹색채권 발행 기반 마련\n- 환경경영 조직 및 관리 체계 구축\n- 환경영향평가 체계 구축\n- 환경경영체계 구축을 통한 환경 리스크 및 성과 관리\n이를 통해 카카오뱅크는 환경을 보호하고 지속 가능한 경영을 추구하고 있습니다.
```

참고문헌

https://wikidocs.net/231233

https://aifactory.space/task/2719/discussion/830

DATA SCIENCE LAB