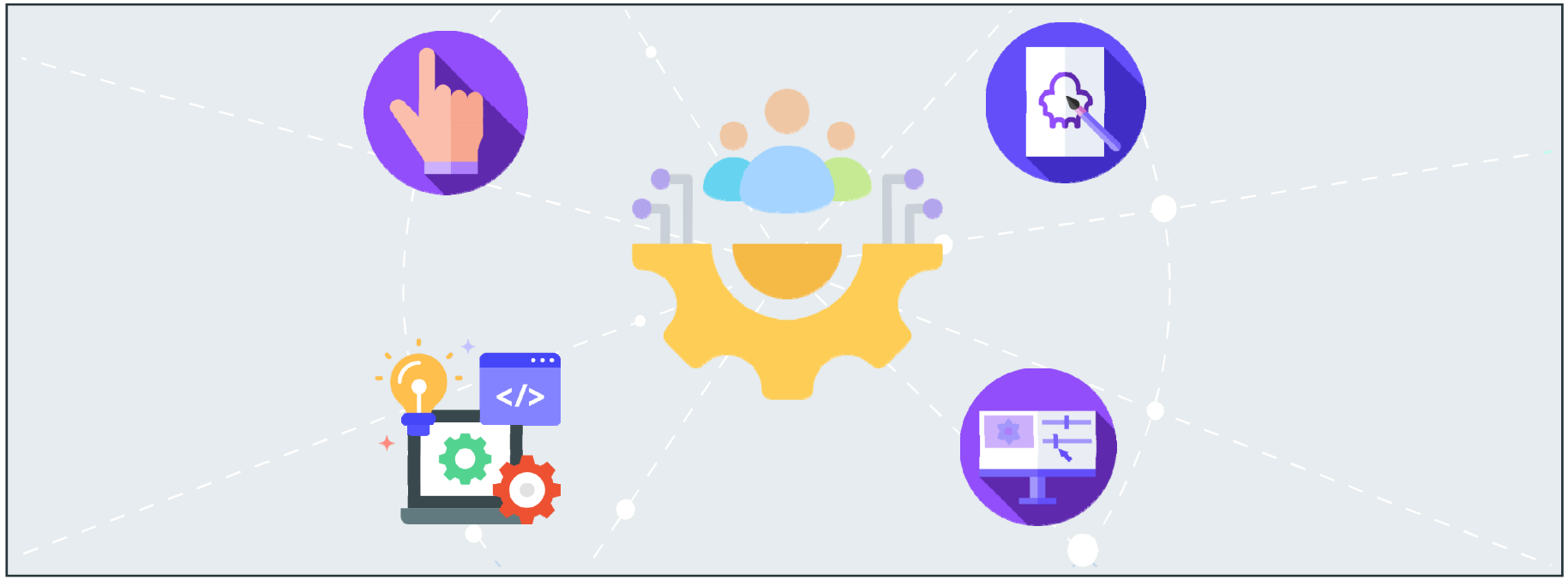
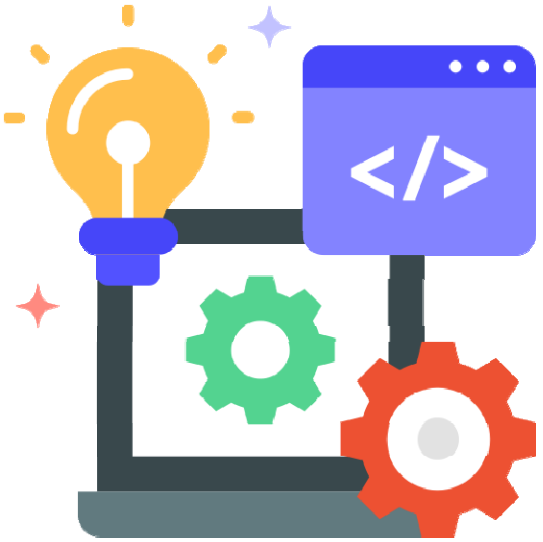


LLM이해와 활용 (RAG)

목차

- Large Language Model 이해
- OpenAI API 주요 기능 활용
- RAG(Retrieval-Augmented Generation)



검색 증강 생성
RAG

소목차

→ RAG

→ Llama Index

→ LangChain

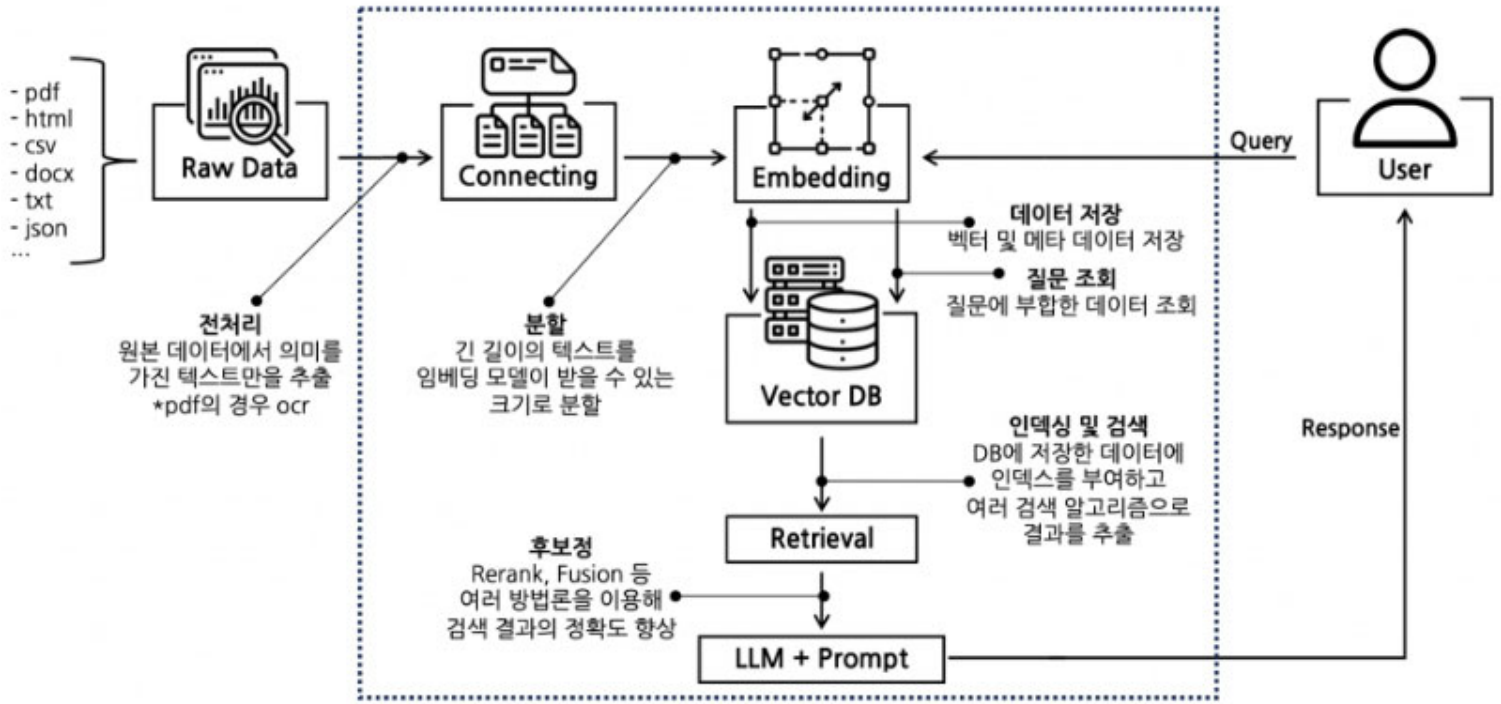
RAG(Retrieval-Augmented Generation)

RAG(Retrieval-Augmented Generation)

➔ 생성(Generation)형 AI 모델에 실시간 정보 검색(Retrieval) 능력을 결합한 접근 방식

➔ 대형 언어 모델(LLM)이 외부 지식 소스와 연계하여 모델의 범용성과 적응력을 유지하면서도 정확하고 신뢰할 수 있는 답변을 생성

➔ 질문 응답 시스템(QA), 지식 검색, 고객 지원 등 다양한 NLP 응용 분야에 활용



RAG(Retrieval-Augmented Generation)

⭘ 챗GPT의 한계와 RAG의 해결책

➔ 챗GPT의 한계

- ▶ 고정된 지식 : ChatGPT(~GPT-3.5)의 지식은 학습 데이터의 기준 시점에 고정되어 있어, 최신 정보를 반영하지 못한다.
- ▶ 한정된 도메인 지식 : 특정 분야의 깊이 있는 전문 지식을 제공하는 데 한계가 있다.
- ▶ 맥락 이해의 한계 : 사용자의 특정 상황이나 맥락을 완벽히 이해하고 반영하기 어렵다.
- ▶ 환각(Hallucination) : 잘못된 정보를 사실인 것처럼 제시하는 경우가 있다.

➔ RAG의 해결책

- ▶ 최신 정보 접근 : 실시간으로 외부 데이터베이스를 검색하여 항상 최신 정보를 반영할 수 있다.
- ▶ 전문 지식 강화 : 특정 도메인의 전문 데이터베이스를 연결하여 깊이 있는 전문 지식을 제공할 수 있다.
- ▶ 맥락 인식 향상 : 사용자의 질문과 관련된 구체적인 정보를 검색하여 더 정확한 맥락 이해가 가능하다.
- ▶ 정보의 신뢰성 향상 : 검색된 실제 데이터를 바탕으로 답변을 생성하므로 환각 현상을 크게 줄일 수 있다.

RAG(Retrieval-Augmented Generation)

RAG 파이프라인

➔ 로딩(Loading) : 다양한 소스(텍스트 파일, PDF, 웹사이트, 데이터베이스, API 등)에서 데이터를 가져와 파이프라인에 입력합니다.

LlamaHub에서 제공하는 다양한 커넥터를 활용할 수 있습니다.

➔ 인덱싱(Indexing) : 데이터를 쿼리 가능한 구조로 변환합니다.

주로 벡터 임베딩을 생성하여 데이터의 의미를 수치화하고, 관련 메타데이터를 함께 저장합니다.

➔ 저장(Storing) : 생성된 인덱스와 메타데이터를 저장하여 재사용할 수 있게 합니다.

➔ 쿼리(Querying) : LLM과 LlamaIndex 데이터 구조를 활용하여 다양한 방식(서브쿼리, 다단계 쿼리, 하이브리드 전략 등)으로 데이터를 검색합니다.

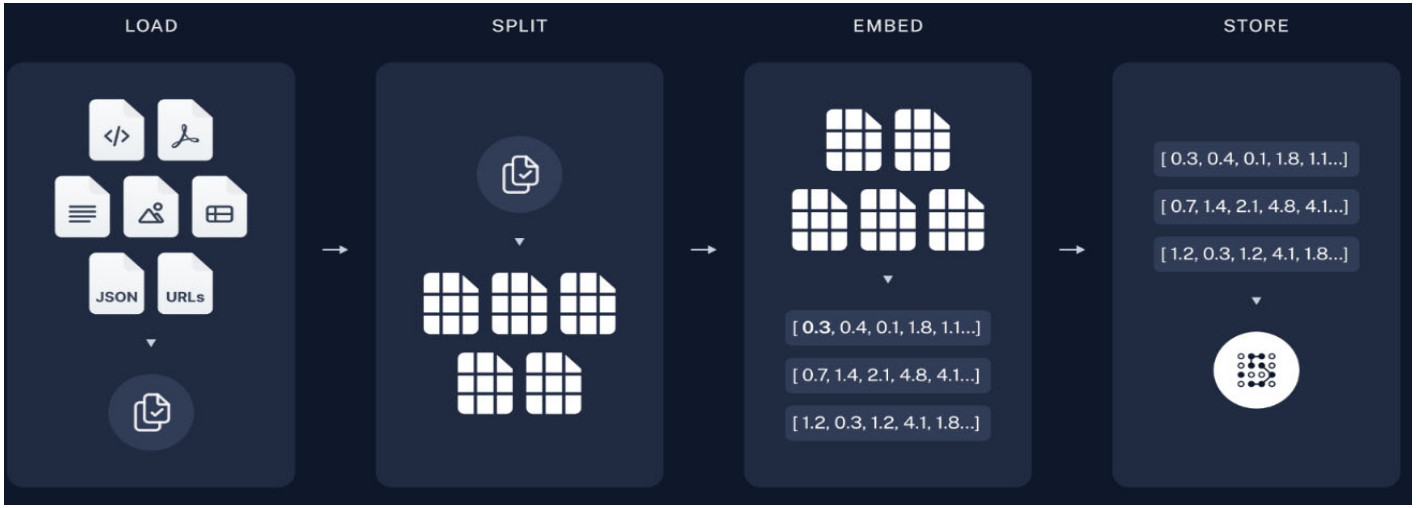
➔ 평가(Evaluation) : 파이프라인의 효과성을 객관적으로 측정합니다. 응답의 정확성, 충실도, 속도 등을 평가합니다.

RAG(Retrieval-Augmented Generation)

RAG 작동 원리

➔ 색인 작업 (Indexing)

- ▶ 다양한 외부 데이터 소스(예: 코드 파일, PDF, 텍스트 문서, 이미지, 스프레드시트, JSON, URLs 등)에서 정보를 추출
- ▶ 로드 (Load) → 분할 (Split) → 임베딩 (Embed) : 분할된 데이터를 벡터 형태로 변환 → 벡터 스토어에 저장 (Store)



RAG(Retrieval-Augmented Generation)

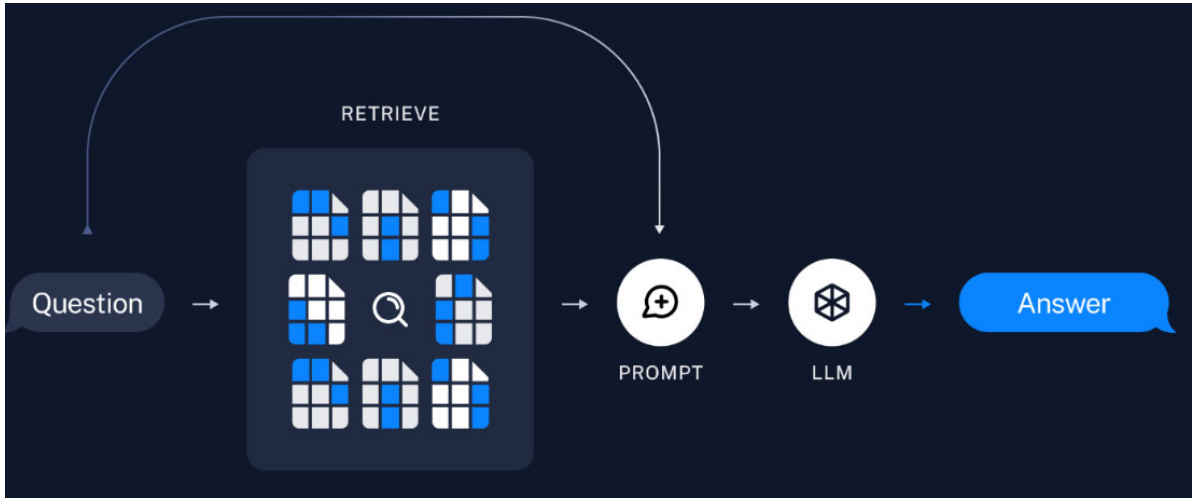
RAG 작동 원리

➔ Retrieval (검색 단계)

- ▶ 질문 분석 (Question Analysis) : 사용자의 질문을 벡터 형태로 변환
- ▶ 벡터 검색(Vector Search) 기법을 사용하여 입력 문장과 관련 있는 문서를 찾음
- ▶ 예: 유사도 검색, TF-IDF, BM25, Dense Vector Search

➔ Augmentation (정보 보강 단계)

- ▶ 검색된 정보를 언어 모델의 입력으로 제공
- ▶ LLM이 기본적으로 이해하는 언어적 지식에 검색된 데이터를 보강



RAG(Retrieval-Augmented Generation)

⭘ RAG 작동 원리

→ Generation (생성 단계)

▶ LLM이 보강된 데이터를 바탕으로 자연스러운 텍스트를 생성

▶ 응답은 단순히 검색된 내용을 복사하는 것이 아니라, 입력과 관련된 문맥을 바탕으로 생성

RAG의 실제 적용 사례	설명
기업 내부 지식관리 시스템	회사의 내부 문서, 정책, 프로세스 등을 RAG 시스템과 연동하여 직원들이 항상 최신의 정확한 정보에 접근할 수 있게 합니다. 예: Slack에 통합된 RAG 기반 챗봇으로 회사 정책에 대한 실시간 질의응답 제공
의료 분야 의사결정 지원 시스템	최신 의학 연구 결과와 환자의 의료 기록을 결합하여 의사의 진단과 치료 결정을 지원합니다. 예: 의사가 환자의 증상을 입력하면 관련된 최신 연구 결과와 유사 사례를 제시하는 RAG 기반 시스템
금융 자문 서비스	실시간 시장 데이터와 개인의 재무 상황을 결합하여 맞춤형 투자 조언을 제공합니다. 예: 사용자의 포트폴리오와 최신 시장 동향을 분석하여 개인화된 투자 전략을 제시하는 AI 금융 어드바이저
고객 서비스 개선	기업의 제품 정보, FAQ, 고객 이력 등을 실시간으로 참조하여 더 정확하고 맞춤화된 고객 응대를 제공

RAG(Retrieval-Augmented Generation)

RAG(Retrieval-Augmented Generation)

➔ RAG 장점

▶ Fine tuning에 비해 시간과 비용이 적게 소요됩니다.

- 외부 데이터베이스를 활용하기 때문에 별도의 학습 데이터를 준비할 필요가 없습니다.

▶ 모델의 일반성을 유지할 수 있습니다.

- 특정 도메인에 국한되지 않고 다양한 분야에 대한 질문에 답변할 수 있습니다.

▶ 답변의 근거를 제시할 수 있습니다.

- 답변과 함께 정보 출처를 제공하여 답변의 신뢰도를 높일 수 있습니다.

▶ 할루시네이션 가능성을 줄일 수 있습니다.

- 외부 데이터를 기반으로 답변을 생성하기 때문에 모델 자체의 편향이나 오류를 줄일 수 있습니다.

Fine tuning은 언어 모델(A)이 사용자의 질문에 정확히 답하기 위해 특정 도메인 지식을 공부하고 학습하여 암기한 상태로 성장시키는 것이라면, RAG는 언어 모델(A)과 도서관 사서가 협업하는 것과 같습니다. 사용자가 질문을 하면, 사서가 도서관의 책 중에서 그 질문에 대한 정보를 담고 있는 책을 찾아낸 후, 언어 모델(A)이 그 책의 내용을 참고하여 질문에 답변하는 것

RAG(Retrieval-Augmented Generation)

Data Load 클래스

➔ 다양한 형태의 데이터 소스에서 정보를 읽어 들여 처리하는 기능을 제공

클래스	설명
WebBaseLoader	웹 기반의 데이터 소스로부터 데이터를 로드 웹 페이지나 API에서 데이터를 수집할 때 사용
TextLoader	일반 텍스트 데이터를 읽어서 LangChain에서 사용할 수 있는 형태로 변환
DirectoryLoader	프로젝트 폴더나 지정된 경로 내의 다수의 파일을 일괄적으로 처리할 때 사용
CSVLoader	테이블 형태의 데이터를 처리하거나 데이터 분석 작업에 주로 사용
PyPDFLoader	PyPDF2 라이브러리를 사용하여 PDF 파일에서 텍스트를 추출
UnstructuredPDFLoader	구조화되지 않은 PDF 파일로부터 데이터를 추출 복잡하거나 일정하지 않은 레이아웃을 가진 PDF에서 텍스트 추출
PyMuPDFLoader	PyMuPDF (Fitz) 라이브러리를 사용하여 PDF 파일로부터 텍스트를 추출
OnlinePDFLoader	웹 URL을 통해 접근 가능한 PDF 문서를 처리할 때 사용
PyPDFDirectoryLoader	디렉토리 내의 모든 PDF 파일을 로드하여 PyPDF2를 이용해 데이터를 추출

RAG(Retrieval-Augmented Generation)

RAG : Load Data

➔ langchain_community.document_loaders 모듈에서 WebBaseLoader 클래스를 사용하여 특정 웹페이지의 텍스트 데이터를 추출하여 Document 객체의 리스트로 변환

```
from langchain_community.document_loaders import WebBaseLoader
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'  # 위키피디아 정책과 지침
loader = WebBaseLoader(url)
docs = loader.load()
print(len(docs))
print(len(docs[0].page_content))
print(docs[0].page_content[5000:6000])
```

RAG(Retrieval-Augmented Generation)

RAG : 텍스트 분할(Text Split)

➔텍스트 분할(Text Split) 도구는 검색 효율성을 높이기 위해 로드한 데이터를 문단, 문장 또는 구 단위의 작은 크기의 단위(chunk)로 분할을 수행합니다.

클래스	설명
CharacterTextSplitter	텍스트를 특정 문자 수에 따라 분할합니다
RecursiveCharacterTextSplitter	재귀적 방법을 사용하여 텍스트를 분할(더 나은 문맥 보존을 시도)
SentenceTextSplitter	텍스트를 문장 단위로 분할합니다.
ParagraphTextSplitter	텍스트를 단락 단위로 분할
SlidingWindowTextSplitter	슬라이딩 윈도우 방식을 사용하여 텍스트를 분할 정해진 길이의 창을 텍스트에 슬라이딩하면서 데이터를 추출하고, 겹치는 영역을 통해 정보의 연속성을 유지
FixedSizeTextSplitter	고정된 크기의 블록으로 텍스트를 나누는 방법각 분할이 동일한 길이를 가지도록 보장하며, 일관된 처리를 요구하는 시나리오에서 효과적

RAG(Retrieval-Augmented Generation)

RAG : 텍스트 분할(Text Split)

➔ LLM 모델이나 API의 입력 크기에 대한 제한이 있기 때문에, 제한에 걸리지 않도록 적정한 크기로 텍스트의 길이를 줄일 필요가 있습니다. (프롬프트가 지나치게 길어질 경우 중요한 정보가 상대적으로 희석되는 문제가 있을 수도 있습니다)

```
# 긴 문장을 최대 1000글자 단위로 분할하며, 200글자는 각 분할마다 겹치게 하여 문맥이 잘려나가지 않고 유지되게 합니다.
from langchain.text_splitter import RecursiveCharacterTextSplitter

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

print(len(splits))
print(splits[10])
print(splits[10].page_content)  #page_content 속성에는 분할된 텍스트 조각이 저장
print(splits[10].medatdata)  #원본 문서의 정보를 포함하는 메타데이터
```

RAG(Retrieval-Augmented Generation)

RAG : 인덱싱(Indexing)

➔ 인덱싱은 검색 시간을 단축시키고, 검색의 정확도를 높이는 데 중요한 역할을 합니다.

➔ LangChain 라이브러리를 사용하여 텍스트를 임베딩으로 변환하고 벡터 저장소에 저장 후 유사성 검색을 수행

클래스	설명
OpenAIEmbeddings	OpenAI의 API를 사용하여 텍스트의 임베딩을 생성 GPT-3 등의 모델을 활용하여 주어진 텍스트에 대한 벡터 표현을 생성
HuggingFaceEmbeddings	Hugging Face의 transformers 라이브러리를 통해 다양한 사전 훈련된 모델(BERT, RoBERTa, DistilBERT 등)로부터 임베딩을 생성합니다.
SpacyEmbeddings	spaCy 라이브러리를 활용하여 텍스트에서 임베딩을 생성 다양한 언어를 지원하며, 텍스트의 문법적, 의미적 특성을 반영한 임베딩을 제공

RAG(Retrieval-Augmented Generation)

RAG : 인덱싱(Indexing)

➔ 외부 데이터를 LLM에 효과적으로 통합하는 프로세스

➔인덱싱은 검색 시간을 단축시키고, 검색의 정확도를 높이는 데 중요한 역할을 합니다.

➔ LangChain 라이브러리를 사용하여 텍스트를 임베딩으로 변환하고 벡터 저장소에 저장 후 유사성 검색을 수행

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

vectorstore = Chroma.from_documents(documents=splits,
                                    embedding=OpenAIEmbeddings())

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

RAG(Retrieval-Augmented Generation)

RAG : 인덱싱(Indexing)

➔ RAG Indexing 구성 요소

- ▶ 데이터 로드: 소스로부터 데이터를 수집
- ▶ 청크 분할: 데이터를 적절한 크기로 분할
- ▶ 임베딩 생성: 텍스트를 벡터로 변환
- ▶ 저장: 벡터 데이터베이스에 저장

➔데이터 전처리 및 청크 분할 최적화

- ▶ 청크 크기 최적화: 2000-3000자 정도가 적절
- ▶ 오버랩 설정: 문맥 유지를 위해 500자 정도 중첩
- ▶ 메타데이터 추가: 출처, 시간 등 관련 정보 포함
- ▶ 품질 관리: 노이즈 제거 및 포맷 정규화

RAG(Retrieval-Augmented Generation)

RAG : 인덱싱(Indexing)

➔ 벡터 임베딩 전략

- ▶ 벡터 임베딩은 RAG의 검색 성능을 좌우하는 핵심 요소입니다
- ▶ 모델 선택: text-embedding-ada-002 등 고성능 모델 활용
- ▶ 차원 최적화: 일반적으로 1536 차원이 표준
- ▶ 정규화: 임베딩 벡터의 품질 유지
- ▶ 배치 처리: 대규모 데이터의 효율적 처리

RAG(Retrieval-Augmented Generation)

RAG : Retrieval & Generation

➔ 사용자의 질문이나 주어진 컨텍스트에 가장 관련된 정보는 사용자의 입력을 바탕으로 쿼리를 생성하고, 인덱싱된 데이터에서 가장 관련성 높은 정보를 LangChain의 retriever()로 검색

➔LLM 모델에 검색 결과와 함께 사용자의 입력을 전달

➔모델은 사전 학습된 지식과 검색 결과를 결합하여 주어진 질문에 가장 적절한 답변을 생성합니다.

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

Question: {question}'''‘
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(model='gpt-4o-mini', temperature=0)
retriever = vectorstore.as_retriever()
….
rag_chain = (
   {'context': retriever | format_docs, 'question': RunnablePassthrough()}
   | prompt
   | model
   | StrOutputParser()
)
rag_chain.invoke("격하 과정에 대해서 설명해주세요.")
```

RAG(Retrieval-Augmented Generation)

RAG Retrievers 최적화

➔Retrieval-Augmented Generation(RAG) 시스템에서 검색기(Retriever)는 사용자의 질문에 대해 관련성 높은 정보를 찾아주는 역할을 합니다.

➔BM25 Retriever

- ▶ 전통적인 키워드 기반 검색 방식으로, 정확한 단어 매칭에 탁월한 성능을 보입니다.
- ▶ 대규모 문서 컬렉션에서도 효율적으로 작동
- ▶ 간단한 알고리즘으로 쉽게 구현하고 튜닝할 수 있다
- ▶ 문맥을 고려하지 않아 관련 있지만 정확한 키워드를 포함하지 않은 문서를 놓칠 수 있습니다.
- ▶ 특정 단어가 자주 등장하는 긴 문서에 편향될 수 있습니다.

➔Vector Store Retriever

- ▶ 텍스트를 고차원 벡터 공간에 매핑하여 의미론적 유사성을 기반으로 검색을 수행
- ▶ 키워드가 정확히 일치하지 않아도 의미적으로 관련된 문서를 검색
- ▶ 최신 자연어 처리 기술의 이점을 활용할 수 있다
- ▶ 임베딩 생성과 저장에 많은 리소스가 필요함
- ▶ 사용된 언어 모델의 품질에 따라 성능이 크게 좌우됨

RAG(Retrieval-Augmented Generation)

RAG Retrievers 최적화

➔MultiQuery Retriever

- ▶ 다양한 형태의 질의를 생성하여 더 많은 관련 문서를 찾아낼 수 있습니다.
- ▶ 사용자 질의의 특정 표현에 덜 민감합니다.
- ▶ 여러 질의를 생성하고 각각에 대해 검색을 수행하므로 계산 비용이 높습니다
- ▶ 관련성이 낮은 문서도 함께 검색될 수 있습니다.

➔Ensemble Retriever

- ▶ 여러 검색 방법의 장점을 결합하여 더 포괄적이고 정확한 검색 결과를 제공
- ▶ 키워드 기반 검색과 의미 기반 검색의 장점을 모두 활용
- ▶ 단일 검색 방법보다 일반적으로 더 나은 성능을 보입니다.
- ▶ 여러 검색기를 관리하고 튜닝하는 것이 더 복잡합니다.
- ▶ 여러 검색기를 실행하고 결과를 결합하는 데 더 많은 컴퓨팅 자원이 필요합니다.

RAG(Retrieval-Augmented Generation)

RAG Retrievers 최적화

➔Maximal Marginal Relevance (MMR)

- ▶ 검색 결과의 다양성과 관련성 사이의 균형을 맞추는 기술
- ▶ 유사한 문서들이 상위 결과를 독점하는 것을 방지
- ▶ 질의와 관련된 문서를 우선적으로 선택
- ▶ 다양성과 관련성 사이의 적절한 균형을 위해 세심한 튜닝이 필요할 수 있습니다.
- ▶ 문서 간 유사도를 계산하는 추가 단계가 필요합니다.

RAG(Retrieval-Augmented Generation)

Rerank

➔검색 결과의 순위를 재조정하는 과정

- ▶ RAG 시스템에서 Rerank는 초기 검색 결과에서 가져온 문서들의 순위를 다시 매기는 역할을 합니다.
- ▶ 사용자의 질문과 가장 관련성 높은 문서들을 상위에 배치하여 LLM이 더 정확한 답변을 생성할 수 있도록 돕습니다.

➔한국어 특화 Reranker

- ▶ 한국어 RAG 시스템의 성능을 높이기 위해서는 한국어에 특화된 Reranker를 사용
- ▶ BAAI/bge-reranker-large 모델을 한국어 데이터로 파인튜닝한 'Dongjin-kr/ko-reranker' 모델을 활용

➔다양한 언어를 지원하는 Reranker

- ▶ 글로벌 서비스를 제공하는 경우, 다양한 언어를 지원하는 Reranker를 사용하는 것이 효과적입니다. 'BAAI/bge-reranker-v2-m3' 모델은 여러 언어를 지원하면서도 빠른 연산 속도를 자랑합니다.

RAG(Retrieval-Augmented Generation)

⭘ RAG기반 지능형 Q&A 웹서비스

➔ LangChain으로 RAG기반 지능형 Q&A 웹서비스 개발

```
pip install langchain langchain_openai chromadb streamlit Wikipedia  langchain_community
```

- ▶ WikipediaLoader : Wikipedia에서 문서를 로드합니다.
- ▶ RecursiveCharacterTextSplitter : 긴 문서를 작은 청크로 분할합니다.
- ▶ OpenAIEmbeddings : 텍스트를 벡터로 변환합니다.
- ▶ Chroma : 벡터 데이터베이스로 사용됩니다.
- ▶ ChatOpenAI : OpenAI의 GPT 모델을 사용합니다.
- ▶ RetrievalQA : 검색-질문 응답 체인을 생성합니다.
- ▶ PromptTemplate : 사용자 정의 프롬프트를 생성합니다.

RAG(Retrieval-Augmented Generation)

⭘ RAG기반 지능형 Q&A 웹서비스

➔ Streamlit : 데이터 앱을 만들 수 있도록 설계된 오픈 소스 라이브러리

- ▶ 복잡한 대시보드를 코딩 없이 몇 줄의 파이썬 코드만으로 생성
- ▶ 데이터 과학자와 엔지니어가 분석 결과를 시각화하고, 데모 앱을 신속하게 프로토타이핑하며, 고객이나 이해관계자와 효과적으로 소통할 수 있는 도구를 제공하는 것이 목적입니다.
- ▶ 복잡한 웹 앱 구조를 몰라도 데이터 앱을 쉽게 구축할 수 있다
- ▶ 코드를 변경하고 저장하는 즉시 앱이 자동으로 업데이트
- ▶ 데이터 표현을 위한 다양한 위젯과 차트를 내장하고 있으며, 라이브러리와의 통합을 통해 더 많은 시각화 옵션을 제공
- ▶ Streamlit Sharing을 통해 GitHub에서 직접 앱을 호스팅하고 공유할 수 있습니다

RAG(Retrieval-Augmented Generation)

RAG기반 지능형 Q&A 웹서비스

```
import streamlit as st
import pandas as pd
import numpy as np

# 제목 추가
st.title('Streamlit Example')
# 데이터 생성
data = pd.DataFrame(
    np.random.randn(50, 3),
    columns=['a', 'b', 'c']
)
# 데이터 표시
st.write("Here's our random data:")
st.write(data)
# 차트 그리기
chart = st.line_chart(data)
```

```
streamlit run your_script.py
```

Llama Index

LlamaIndex 란?

➔ LLM에서 학습되지 않은 데이터를 참조해서 질의응답 채팅 AI를 쉽게 만들 수 있는 오픈 소스 라이브러리

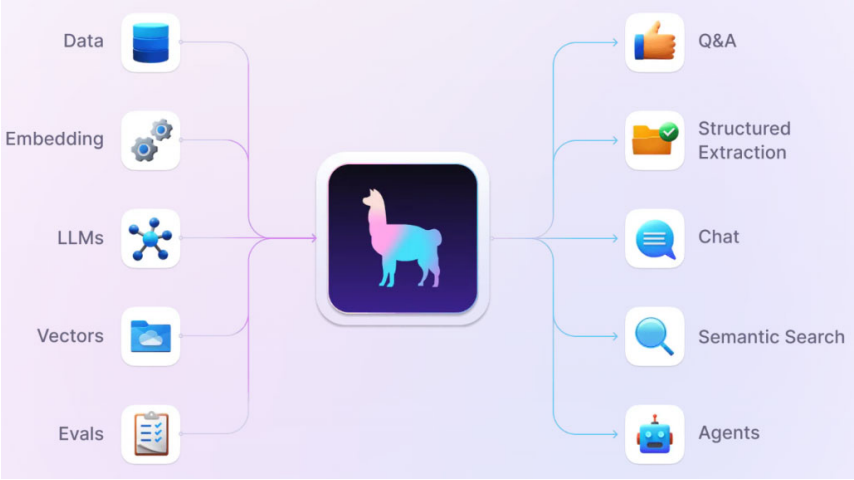
➔ RAG(Retrieval-Augmented Generation) 작업 흐름을 간단한 Python 코드로 구현할 수 있게 해주는 강력한 오픈소스 라이브러리

➔ 외부 데이터를 LLM에 주입해 더 정확하고 최신의 응답을 생성할 수 있게 해주는 기술

➔ 내부적으로 LangChain 라이브러리를 사용

➔ https://github.com/run-llama/llama_index

➔ https://docs.llamaindex.ai/en/stable/examples/data_connectors/simple_directory_reader//



Llama Index

LlamaIndex 란?

➔RAG 파이프라인을 직접 구축하려면 데이터 로딩부터 인덱싱, 검색, 프롬프트 생성 등 복잡한 과정이 필요합니다.

➔ LlamaIndex는 RAG(Retrieval-Augmented Generation) 작업흐름을 간단한 Python 코드로 구현할 수 있게 해주는 강력한 오픈소스 라이브러리입니다. RAG는 외부 데이터를 LLM에 주입해 더 정확하고 최신의 응답을 생성할 수 있게 해주는 기술로, LlamaIndex는 이러한 RAG 파이프라인을 쉽고 효과적으로 구현할 수 있게 해줍니다

➔LlamaIndex는 개인 데이터를 가져 오고 구조화하는 도구를 제공하며 데이터에 대한 고급 검색 / 쿼리 인터페이스를 생성하고 외부 응용 프레임 워크와 쉽게 통합 할 수 있도록 지원하여 ChatGPT와 함께 작동합니다.

➔ 응용 분야 :

- ▶ 고객 지원을 위한 고급 챗봇 생성, 연구 및 학술계를위한 도메인 특정 응답 제공, 보건 전문가를위한 상세한 의료 정보 제공 등

➔ LlamaIndex를 ChatGPT와 함께 구현하려면 데이터 수집, 적재, 구조화, 쿼리 및 통합과 같은 여러 단계가 필요합니다.

➔다른 모델 및 프레임 워크와 함께 사용할 수 있도록 유연하게 설계된 솔루션

Llama Index

LlamaIndex 주요 기능

➔ 다양한 데이터 소스 지원

- ▶ 텍스트, PDF, ePub, 워드, 파워포인트, 오디오를 비롯한 다양한 파일 형식과 트위터, 슬랙, 위키피디아 등 웹 서비스를 자체 데이터로 지정 가능

➔ 로드한 데이터를 벡터 임베딩으로 변환하고 효율적인 검색을 위해 인덱싱

➔ 사용자의 쿼리에 대해 가장 관련성이 높은 문서나 데이터 조각을 검색하는 다양한 검색 알고리즘을 지원

➔ 검색된 관련 문서를 바탕으로 LLM을 활용하여 사용자 쿼리에 대한 정확하고 상세한 응답을 생성

➔ LlamaIndex는 모듈화된 구조로 설계

- ▶ 각 컴포넌트를 필요에 따라 커스터마이즈하거나 교체할 수 있습니다.

➔ 다양한 벡터 데이터베이스, 임베딩 모델, LLM 등을 지원

➔ 쿼리 최적화

- ▶ 복잡한 쿼리를 자동으로 분해하고 최적화하여 더 정확한 응답을 생성

➔ 멀티모달 데이터 처리

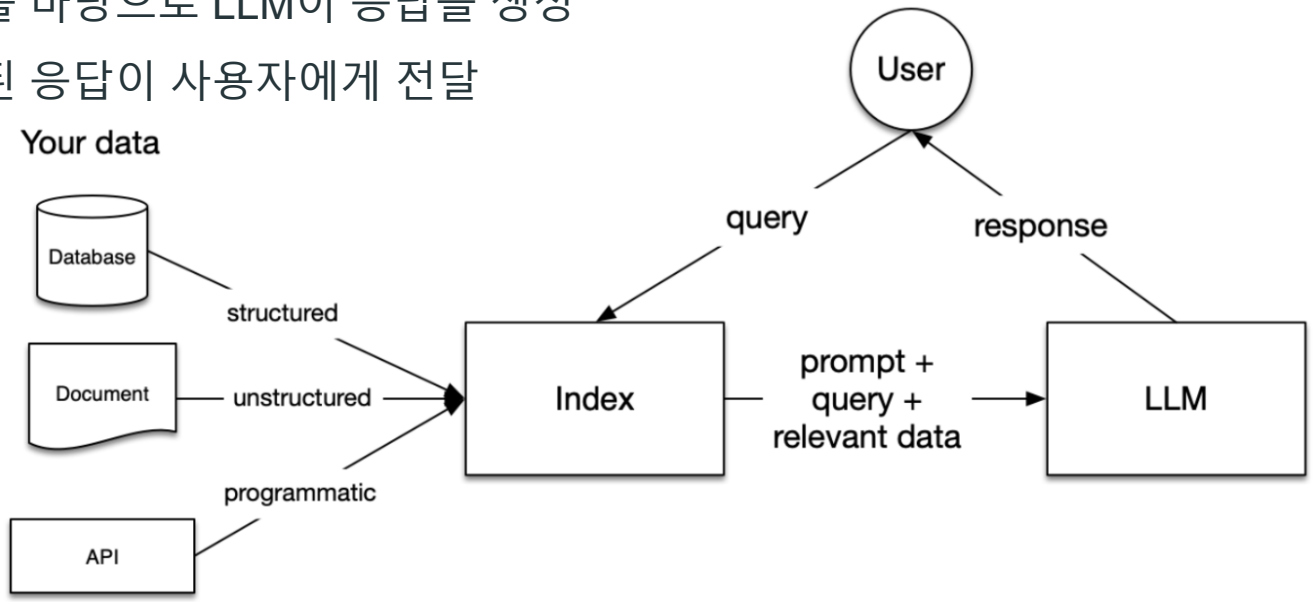
- ▶ 텍스트뿐만 아니라 이미지, 오디오 등 다양한 형태의 데이터를 처리할 수 있는 기능을 제공

Llama Index

LlamaIndex 아키텍처

➔다양한 데이터 소스 지원

1. 데이터는 "인덱스"라는 형태로 변환되어 쿼리에 사용될 수 있도록 준비
2. 사용자가 질문을 입력하면, 쿼리는 인덱스에 전달
3. 인덱스는 사용자의 쿼리와 가장 관련성 높은 데이터를 필터링
4. 필터링된 관련 데이터, 원래 쿼리, 적절한 프롬프트가 LLM에 전달
5. 정보를 바탕으로 LLM이 응답을 생성
6. 생성된 응답이 사용자에게 전달



Llama Index

LlamaIndex

➔ Loading

- ▶ 텍스트 파일, PDF 파일, 웹 사이트, 데이터베이스, API 등의 데이터를 파이프라인에 넣는 것이다.
- ▶ LlamaHub에 다양한 connector가 있다.

➔ Document

- ▶ 데이터 소스에 대한 컨테이너이다.

➔Node

- ▶ 라마인덱스 데이터의 한 단위이고 Document의 chunk이다.
- ▶ Node에는 메타데이터가 포함된다.

➔Connectors

- ▶ Reader라고도 불린다.
- ▶ 데이터를 여러 소스로부터 받아서 Document와 Node를 만든다.

Llama Index

LlamaIndex

➔ Indexing

- ▶ 데이터를 쿼리할 수 있는 자료 구조를 생성하는 것이다.
- ▶ LLM에서는 주로 벡터 임베딩을 의미한다.

➔Indexes

- ▶ 데이터를 ingest하고 나면 retrieve 하기 쉬운 구조로 만들어야한다.
- ▶ 주로 벡터 임베딩을 생성한다.
- ▶ Indexes는 데이터에 대한 메타데이터도 포함할 수 있다.

➔Embeddings

- ▶ LLM은 숫자로 표현된 데이터인 임베딩을 만들어낸다.

➔Storing

- ▶ 데이터가 인덱싱되면 다시 인덱싱하지 않아도 되도록 그 인덱스를 저장한다.

➔Querying

Llama Index

LlamaIndex 데이터 로드 및 인덱스 생성

```
pip install llama-index
```

```
# 로컬 파일 시스템에서 텍스트 데이터 로드, 질문에 대한 답변을 제공하는 시스템을 구현
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

documents = SimpleDirectoryReader("data").load_data()
index = VectorStoreIndex.from_documents(documents)
#쿼리 실행
query_engine = index.as_query_engine()
response = query_engine.query("원하는 질문을 입력하세요")
print(response)
```

➔ 문서 로드

➔ 인덱스 생성

➔ 쿼리 엔진 생성

➔ 질의 응답

➔ 인덱스 저장 및 로드

https://docs.llamaindex.ai/en/stable/module_guides/indexing/vector_store_index/

Llama Index

LlamaIndex

➔ LlamaIndex를 활용한 Chatbot 개발

- ▶ 사용할 LLM api를 통해서 Custom Model 클래스를 생성
- ▶ 사용할 embedding api를 통해서 Custom Embed 클래스를 생성
- ▶ 임베딩 값을 저장할 vector db를 세팅
- ▶ 임베딩할 문서가 임베딩 가능한 사이즈보다 크다면 parser를 사용하여 나눠준다.
- ▶ 알맞은 사이즈로 분할된 문서를 임베딩한다.
- ▶ 필요하다면 노드에 metadata를 추가한다. 표 데이터의 경우는 각 column이 어떤 의미를 갖는지를 metadata에 추가할 수 있다. 이 table_summary라는 metadata는 임베딩은 되지 않지만 LLM에게만 전달되도록 설정할 수 있다.
- ▶ 사용자 쿼리가 들어오면 임베딩을 한다.
- ▶ Vector DB에서 similarity score가 높은 노드들을 뽑는다.
- ▶ 필요에 따라서 post processor를 통해서 retrieved 노드들을 가공한다.
- ▶ Prompt template을 통해서 LLM에게 요청을 보낸다.
- ▶ Retrieved nodes 수가 많아서 LLM이 한 번에 처리할 수 있는 데이터의 크기보다 크다면 refine 과정을 통해서 답을 만들어간다.

Llama Index

Ollama

➔ 대규모 언어 모델(LLMs)을 로컬 환경에서 설정하고 사용할 수 있도록 지원하는 도구

- ▶ 사용자가 대규모 언어 모델을 손쉽게 로컬 컴퓨터에 설치하고 설정할 수 있도록 도와줍니다.
- ▶ 현재 OSX와 Linux에서 사용 가능하며, Windows 사용자는 WSL 2를 통해 간접적으로 이용할 수 있습니다
- ▶ 로컬 환경에서 LLM을 운영하면서 필요한 리소스 관리, 성능 최적화, 업데이트 관리 등을 수행할 수 있게 지원할 수 있습니다
- ▶ 로컬에서 모델을 직접 관리함으로써 개발자와 연구자들은 실험 및 개발 과정에서 더 빠르고 유연하게 작업을 진행할 수 있습니다.

https://docs.llamaindex.ai/en/stable/getting_started/starter_example_local/

Llama Index

llamahub

➔라마인덱스에서 제공하는 데이터 커넥터를 이용하면 다양한 데이터(PDF, ePub, 워드, 파워포인트, 오디오 등)과 웹 서비스(트위터, 슬랙, 위키백과 등)를 문서의 데이터 소스로 활용할 수 있습니다

- ▶ 데이터 로더와 에이전트 도구를 혼합하고 조합하여 맞춤형 RAG 앱을 만들거나

https://llamahub.ai/

Vector Store

Vector Store

➔ 데이터를 임베딩(벡터 표현)으로 저장해서 벡터 공간 내에서 빠른 검색을 구현하기 위한 DB

- ▶ 임베딩 벡터는 텍스트, 이미지, 소리 등 다양한 형태의 데이터를 벡터 공간에 매핑한 것으로, 데이터의 의미적, 시각적, 오디오적 특성을 수치적으로 표현
- ▶ 코사인 유사도, 유클리드 거리, 맨해튼 거리 등 다양한 유사도 측정 방법을 제공

➔ 패키지로 제공

- ▶ Faiss (https://github.com/facebookresearch/faiss)
- ▶ Qdrant (https://qdrant.tech)
- ▶ Chroma (https://docs.trychroma.com)
- ▶ Milvus (https://milvus.io)

➔ 클라우드 서비스로 제공

- ▶ Pinecone(https://www.pinecone.io)
- ▶ Weaviate (https://weaviate.io)

Vector Store

Vector Store

➔ 비정형 데이터 처리 : 텍스트, 이미지 등 다양한 형태의 비정형 데이터를 효율적으로 저장하고 검색할 수 있습니다.

➔고차원 데이터 처리 : 수백에서 수천 차원의 고차원 벡터 데이터를 빠르게 처리할 수 있습니다.

➔유사도 기반 검색 : 벡터 간 유사도를 계산하여 가장 유사한 데이터를 빠르게 찾아낼 수 있습니다.

➔LLM 성능 향상 : 방대한 양의 정보를 효율적으로 검색하여 LLM에 제공함으로써 성능을 크게 향상시킬 수 있습니다.

VectorDB	설명
Faiss	Facebook AI Research에서 개발한 오픈소스 라이브러리로, 대규모 데이터셋에서 효율적인 유사도 검색을 지원
Annoy	Spotify에서 개발한 라이브러리로, 근사 최근접 이웃(Approximate Nearest Neighbor) 검색에 특화
Chroma	오픈소스 벡터 데이터베이스로, 로컬 환경에서 쉽게 사용할 수 있습니다
Pinecone	클라우드 기반의 벡터 데이터베이스 서비스로, 대규모 데이터 처리에 적합
Qdrant	비동기 작업을 지원하는 벡터 데이터베이스로, 실시간 애플리케이션에 적합

Vector Store

Chroma Vector Store

➔ Chroma.from_texts 메소드를 사용하여 분할된 텍스트들을 임베딩하고, Chroma 벡터 저장소에 임베딩을 저장합니다.

➔ 저장소는 collection_name으로 구분

➔ collection_metadata에서 'hnsw:space': 'cosine'을 설정하여 유사도 계산에 코사인 유사도를 사용

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

Vector Store

MMR (Maximum marginal relevance search)

➔ 최대 한계 관련성(Maximum Marginal Relevance, MMR) 검색 방식은 유사성과 다양성의 균형을 맞추어 검색 결과의 품질을 향상시키는 알고리즘

➔ 검색 쿼리에 대한 문서들의 관련성을 최대화하는 동시에, 검색된 문서들 사이의 중복성을 최소화하여, 사용자에게 다양하고 풍부한 정보를 제공

- query: 사용자로부터 입력 받은 검색 쿼리입니다.
- k: 최종적으로 선택할 문서의 수 (반환할 문서의 총 개수를 결정)
- fetch_k : MMR 알고리즘을 수행할 때 고려할 상위 문서의 수 (초기 후보 문서 집합의 크기를 의미하며, 이 중에서 MMR에 의해 최종 문서가 k개 만큼 선택됩니다.)
- lambda_mult : 쿼리와의 유사성과 선택된 문서 간의 다양성 사이의 균형을 조절합니다. (λ=1) 은 유사성만 고려하며, (λ=0)은 다양성만을 최대화합니다.

Vector Store

FAISS (Facebook AI Similarity Search)

➔ Facebook AI Research에서 개발한 라이브러리

➔ 고차원 벡터 간의 유사성 검색을 빠르고 효율적으로 수행할 수 있도록 설계

➔ CPU와 GPU 모두에서 사용 가능

➔ 최적화된 인덱싱 구조를 사용하여 수백만 또는 수십억 개의 벡터를 처리할 수 있으며, 벡터 간의 유사성 검색을 매우 빠르게 수행

➔다양한 인덱싱 메커니즘을 제공

- ▶ 정확한 검색을 위한 Flat 인덱스
- ▶ 공간 효율성을 높이는 IVF (Inverted File Indexing) 인덱스
- ▶ 양자화를 사용하여 저장 공간을 절약하는 PQ (Product Quantization) 인덱스

➔단일 머신에서와 클러스터 환경에서도 확장 가능하도록 설계

* 추천 시스템 : 사용자나 아이템의 특성을 벡터로 표현한 후, 가장 유사한 아이템을 빠르게 찾아내 추천합니다.
이미지 검색 : 이미지를 대표하는 특성 벡터를 인덱싱하여, 비슷한 이미지를 신속하게 검색
클러스터링 : 대량의 데이터를 효과적으로 클러스터링하기 위해 사용됩니다.
자연어 처리 : 문서나 단어의 임베딩 벡터를 통해 텍스트의 유사도를 판별하거나 관련 문서를 검색합니다.

Vector Store

FAISS (Facebook AI Similarity Search)

➔ https://github.com/facebookresearch/faiss

➔ Index검색 알고리즘

알고리즘	설명
IndexFlatL2	L2 거리(유클리디안 거리)를 사용하여 쿼리 벡터와 데이터베이스 내 모든 벡터 간의 거리를 계산하는 브루트-포스 방식의 검색 인덱스 모든 벡터를 단순하게 저장하고, 쿼리가 주어질 때 모든 벡터와의 거리를 계산하여 가장 가까운 이웃을 찾습니다. 정확도는 매우 높지만, 벡터의 수가 많을 때는 계산 비용이 크게 증가합니다.
IndexFlatIP	내적(dot product)을 기반으로 검색을 수행하는 인덱스 내적은 코사인 유사도와 관련이 깊으며, 벡터 간의 각도 또는 방향성을 비교하는 데 유용 추천 시스템이나 텍스트 임베딩과 같이 벡터 간의 유사도를 평가할 필요가 있을 때 사용
IndexIVFFlat	데이터베이스를 여러 개의 클러스터로 나누고 각 클러스터에 대한 인덱스를 구성합니다. 쿼리 벡터는 가장 가까운 클러스터 몇 개를 빠르게 찾고, 그 안에서만 보다 정밀한 검색을 수행합니다. 이 방식은 대규모 데이터셋에 대해 높은 검색 속도를 제공하면서도 적당한 정확도를 유지할 수 있습니다.

Vector Store

FAISS (Facebook AI Similarity Search)

➔Index검색 알고리즘

알고리즘	설명
IndexIVFPQ	IndexIVFFlat의 개념을 확장, 데이터를 더욱 효율적으로 저장하기 위해 Product Quantization을 적용 메모리 사용량을 크게 줄이면서도 빠른 검색 속도를 제공하고, 대규모 데이터 셋에 적합 (Inverted File with Product Quantization)
IndexLSH	고차원 데이터를 저차원 공간으로 해싱하여, 유사한 데이터 포인트가 같은 해시 버킷에 떨어지도록 합니다 매우 빠른 검색 속도를 제공하지만, 정확도는 다른 방법보다 낮을 수 있습니다. (Locality-Sensitive Hashing)
IndexHNSW	그래프 기반 검색 알고리즘 (Hierarchical Navigable Small World) 다수의 계층에서 효율적인 경로를 통해 빠르게 근접 이웃을 찾습니다. 높은 차원의 데이터에서도 우수한 검색 성능과 정확도를 제공
IndexPQ	벡터를 여러 부분으로 나누고 각 부분을 양자화하여 저장합니다. (Product Quantization)
IndexSQ	벡터의 각 차원을 독립적으로 양자화합니다. (Scalar Quantizer) 차원별로 양자화를 수행하여 더욱 세밀한 제어가 가능합니다.

Vector Store

Faiss Vector DB 초기화

```
pip install faiss-cpu  # CPU 버전
pip install faiss-gpu  # GPU 버전
```

```
import faiss
import numpy as np

dimension = 64  # 벡터의 차원
num_vectors = 1000  # 벡터의 수

# 벡터 데이터 생성
db_vectors = np.random.random((num_vectors, dimension)).astype('float32')
# 인덱스 생성
index = faiss.IndexFlatL2(dimension)
index.add(db_vectors)
# 쿼리 벡터 생성
query_vectors = np.random.random((5, dimension)).astype('float32')
# 가장 가까운 이웃 검색
D, I = index.search(query_vectors, k=5)
print('Distances:', D)
print('Indices:', I)
```

Vector Store

Faiss Vector DB 기반 llama_index 실습

```
pip install llama-index-vector-stores-faiss
```

```
import faiss
# faiss의 인덱스 생성
faiss_index = faiss.IndexFlatL2(1536)

from llama_index.vector_stores.faiss import FaissVectorStore

#인덱스 생
vector_store = FaissVectorStore(faiss_index=faiss_index)
storage_context = StorageContext.from_defaults(vector_store=vector_store)
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context
)
# 쿼리 엔진 생성
query_engine = index.as_query_engine()
# 질의응답
print(query_engine.query("미코의 소꿉친구 이름은?"))
```

https://docs.llamaindex.ai/en/stable/examples/vector_stores/FaissIndexDemo/