
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

CH10 검색기 (Retriever)

- 벡터스토어 저장 단계 = **RAG** 의 **5번째 단계**
  - 저장된 벡터 데이터베이스에서 **사용자의 질문과 관련된 문서를 검색하는 과정**
  - 목표: 사용자 질문에 **가장 적합한 정보를 신속하게 찾아내는 것**
  - RAG 시스템의 전반적인 성능과 직결되는 매우 중요한 과정

- 검색기의 **필요성**
  - **정확한 정보 제공**
    - 사용자의 질문과 **가장 관련성 높은 정보를 검색** → 시스템이 **정확하고 유용한 답변을 생성할 수 있도록 함**
    - 이 과정이 효과적으로 이루어지지 않으면, 결과적으로 제공되는 **답변의 품질이 떨어질 수 있음**
  - **응답 시간 단축**
    - 효율적인 검색 알고리즘 사용 → 데이터베이스에서 적절한 **정보를 빠르게 검색** → 전체적인 **시스템 응답 시간 단축**
    - 사용자 경험 향상에 직접적인 영향을 미침
  - **최적화**
    - 효과적인 검색 과정을 통해 **필요한 정보만을 추출** → 시스템 자원의 사용최적화, **불필요한 데이터 처리 ↓**

- **동작 방식**
  - **질문의 벡터화**
    - 사용자의 질문을 벡터 형태로 변환
    - 임베딩 단계와 유사한 기술을 사용해 진행

        - 반환된 질문 벡터 = 후속 검색 작업의 기준점으로 사용됨
    - **`벡터 유사성 비교`**
        - 저장된 문서 벡터들과 질문 벡터 사이의 **`유사성을 계산`**
            - *유사성 계산*: 주로 **`cosine similarity`**, **`MMR`** (`Max Marginal Relevance`) 등
    - **`상위 문서 선정`**
        - 계산된 유사성 점수를 기준으로 **`상위 N개의 가장 관련성 높은 문서를 선정`**
        - 이 문서들은 다음 단계에서 사용자의 질문에 대한 답변을 생성하는 데 사용
    - **`문서 정보 반환`**
        - 선정된 문서들의 정보 → 다음 단계(`프롬프트 생성`)로 전달
        - **`문서의 내용`**, **`위치`**, **`메타데이터`** **등이 포함될 수 있음**

- **`검색기의 중요성`**
    - `RAG` 시스템에서 **`정보 검색의 질을 결정하는 핵심적인 역할`**
        - 효율적인 검색기 없이는 대규모 데이터베이스에서 관련 정보를 신속하고 정확하게 찾아내는 것 = 매우 어려움
    - **`사용자의 질문에 대한 적절한 컨텍스트 제공`** **→ 언어 모델이 보다** **`정확한 답변을 생성`****할 수 있도록 도움 →** **`RAG`** **시스템의 전반적인 효율성과 사용자 만족도에 직접적인 영향을 미침**

- **`주요 두 가지 방법`**
    - **`Sparse Retriever`**
        - 문서, 질문(`query`)를 이산적인 **`키워드 벡터로 변환하여 처리`**
        - 주로 텀 빈도-역문서 빈도(`TF-IDF`)나 `BM25` 같은 전통적인 정보 검색 기법 사용

        - **`TF-IDF`** (`Term Frequency-Inverse Document Frequency`)
            - 단어가 문서에 나타나는 빈도와 그 단어가 몇 개의 문서에서 나타나는지를 반영하여 단어의 중요도를 계산
            - 자주 나타나면서도 문서 집합 전체에서 드물게 나타나는 단어가 높은 가중치를 받음
        - **`BM25`**
            - `TF-IDF`를 개선한 모델 → 문서의 길이를 고려하여 검색 정확도를 향상시킴
            - 긴 문서와 짧은 문서 간의 가중치 조정 → 단어 빈도의 영향을 상대적으로 조절

        - 특징
            - 각 단어의 존재 여부만을 고려 → 계산 비용이 낮고, 구현이 간단
            - 단어의 의미적 연관성을 고려하지 않음 → **검색 결과의 품질이 키워드의 선택에 크게 의존**

- **Dense Retriever**
    - 최신 딥러닝 기법을 사용 → 문서, `query`를 연속적인 고차원 벡터로 인코딩
    - 문서의 의미적 내용을 보다 풍부하게 표현 가능 → **키워드가 완벽히 일치하지 않더라고 의미적으로 관련된 문서를 검색 가능**

    - 벡터 공간에서의 거리 (예시: `코사인 유사도`)를 사용 → **쿼리와 가장 관련성이 높은 문서를 찾음**
        - 언어의 뉘앙스와 문맥을 이해하는 데 유리
        - **복잡한 쿼리에 대해 더 정확한 검색 결과를 제공 가능**

- **차이점**
    - **표현 방식**
        - `Sparse Retriever`: 이산적인 키워드 기반의 표현을 사용
        - `Dense Retriever`: 연속적인 벡터 공간에서의 의미적 표현 사용
    - **의미적 처리 능력**
        - `Dense Retriever`: 문맥, 의미 더 깊이 파악 → 키워드가 정확히 일치하지 않아도 관련 문서 검색 가능
        - `Sparse Retiever`: 의미적 뉘앙스 덜 반영함
    - **적용 범위**
        - `Dense Retriever`: 복합한 질문, 자연어 쿼리에 더 적합
        - `Sparse Retriever`: 간단, 명확한 키워드 검색에 더 유용

- **코드**
    - **Dense Retriever**

```
from langchain_community.vectorstores import FAISS

# 단계 4: DB 생성(Create DB) 및 저장
# 벡터스토어 생성하기
```

```
vectorstore = FAISS.from_documents(documents=split_documents, embedding=embe

# 단계 5: Dense Retriever 생성
# 문서에 포함되어 있는 정보를 검색하고 생성하기
faiss_retriever = vectorstore.as_retriever()
```

- ◦ **`Sparse Retriever`**

```
from langchain_community.retrievers import BM25Retriever

# 단계 5: Sparse Retriever 생성
# 문서에 포함되어 있는 정보를 검색하고 생성하기
bm25_retriever = BM25Retriever.from_documents(split_documents)
```

- **참고**
  - ***`벡터저장소 지원 검색기`***
  - ***`LangChain Retriever`***

---

1. **`VectorStore-backed Retriever`**

1) **`벡터스토어 기반 검색기`**

- `Vector Store`를 사용해 문서를 검색하는 `Retriever`
- `Vector Store`에 구현된 **`유사도 검색`** (**`similarity search`**), **`MMR`** 같은 검색 메서드를 사용 → `Vector Store` 내의 텍스트를 쿼리

---

2) **`설정`**

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv

# API 키 정보 로드
load_dotenv()                                   # True
```

```
True
```

```
from langsmith import Client
from langsmith import traceable

import os

# LangSmith 환경 변수 확인

print("\n--- LangSmith 환경 변수 확인 ---")
langchain_tracing_v2 = os.getenv('LANGCHAIN_TRACING_V2')
langchain_project = os.getenv('LANGCHAIN_PROJECT')
langchain_api_key_status = "설정됨" if os.getenv('LANGCHAIN_API_KEY') else "설정되지

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langchai
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='{langchain_tracing_v2}
    print(f"✅ LangSmith 프로젝트: '{langchain_project}'")
    print(f"✅ LangSmith API Key: {langchain_api_key_status}")
    print("  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.")
else:
    print("❌ LangSmith 추적이 완전히 활성화되지 않았습니다. 다음을 확인하세요:")
    if langchain_tracing_v2 != "true":
        print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있지 않습니다 (현재: '{langc
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

- 셀 출력

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
-> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

```
# 허깅페이스 임베딩 모델
import gc
```

```
import numpy as np
import time
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_core.documents import Document
import warnings

# 경고 무시
warnings.filterwarnings("ignore")

embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
    )

# 임베딩
embeddings = embeddings

# 임베딩 차원 크기를 계산
dimension_size = len(embeddings.embed_query("hello world"))
print(dimension_size)                                    # 384
```

```
384
```

```
from langchain_community.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_text_splitters import CharacterTextSplitter
from langchain_community.document_loaders import TextLoader

# TextLoader를 사용하여 파일 로드하기
loader = TextLoader("../10_Retriever/data/appendix-keywords.txt")

# 문서 로드하기
documents = loader.load()

# 문자 기반으로 텍스트를 분할하는 CharacterTextSplitter를 생성하기
text_splitter = CharacterTextSplitter(
    chunk_size=300,                          # 청크 크기 = 300
    chunk_overlap=0                          # 청크 간 중복 없음
    )

# 로드된 문서 분할하기
split_docs = text_splitter.split_documents(documents)

# 허깅페이스 임베딩 모델로 임베딩 생성하기
embeddings = embeddings

# 분할된 텍스트와 임베딩을 사용하여 FAISS 벡터 데이터베이스 생성하기
db = FAISS.from_documents(split_docs, embeddings)
```

---

3) `VectorStore에서 VectorStoreRetriever 초기화` (`as_retriever`)

- **`as_retriever`** 메서드: `VectorStore` 객체를 기반으로 `VectorStoreRetriever` 초기화 → 반환

- **매개변수**
  - **`kwargs`**: 검색 함수에 전달할 키워드 인자
  - **`search_type`**: 검색 유형
    - `similarity`
    - `MMR`
    - `similarity_score_threshold`
  - **`search_kwargs`**: 추가 검색 옵션
    - **`k`**: 반환할 문서 수 *(`기본값 = 4`)*
    - **`score_threshold`**: `similarity_score_threshold` 검색의 최소 유사도 임계값
    - **`fetch_k`**: `MMR` 알고리즘에 전달할 문서 수 *(`기본값 = 20`)*
    - **`lambda_mult`**: `MMR` 결과의 다양성 조절 *(`0~1 사이`, `기본값=0.5`)*
    - **`filter`**: 문서 메타데이터 기반 필터링

- **반환값**
  - **`VectorStoreRetriever`**: 초기화된 `VectorStoreRetriever` 객체

- **참고**
  - 다양한 검색 전략 구현 가능: `유사도`, `MMR`, `임계값 기반`
  - `MMR` (`Maximal Marginal Relevance`) 알고리즘으로 검색 결과의 다양성 조절 가능
  - 메타데이터 필터링으로 특정 조건의 문서만 검색 가능
  - `tag` 매개변수를 통해 검색기에 태그 추가 가능

- **주의사항**
  - **`search_type`**, **`search_kwargs`** 적절한 조합 필요
  - `MMR` 사용 시: **`fetch_k`**, **`k`** 값의 균형 조절 필요
  - **`score_threshold`** 설정 시 너무 높은 값은 검색 결과가 없을 수 있음
  - 필터 사용 시 데이터 셋의 메타데이터 구조를 정확히 파악할 필요가 있음
  - **`lambda_mult`** 값이 `0`에 가까울수록 다양성이 높아지고, `1`에 가까울수록 유사성이 높아짐

```
# 데이터베이스를 검색기로 사용하기 위해 retriever 변수에 할당

retriever = db.as_retriever()
```

---

4) `Retriever의 invoke()`

- **`invoke`** 메서드
  - `Retriever`의 주요 진입점 → 관련 문서를 점색하는 데 사용
  - 동기적으로 `Retriever`를 호출 → 주어진 쿼리에 대한 관련 문서 반환

- **`매개변수`**
  - **`input`**: 검색 쿼리 문자열
  - **`config`**: `Retriever` 구성 (`Optional`[`RunnableConfig`])
  - **`kwargs`**: `Retriever`에 전달할 추가 인자

- **`반환값`**
  - **`List`** **[`Document`]**: 관련 문서 목록

```
# 관련 문서를 검색해보기
docs = retriever.invoke("임베딩(Embedding)은 무엇인가요?")

# 출력해보기
for doc in docs:
    print(doc.page_content)
    print("============================================================")
```

- 셀 출력 (0.5s)

```
정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽
예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.
연관키워드: 데이터 교환, 웹 개발, API

Transformer
============================================================
정의: 토큰은 텍스트를 더 작은 단위로 분할하는 것을 의미합니다. 이는 일반적으로 단어, 문장, 또는 구절일
예시: 문장 "나는 학교에 간다"를 "나는", "학교에", "간다"로 분할합니다.
연관키워드: 토큰화, 자연어 처리, 구문 분석

Tokenizer
============================================================
정의: 토크나이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리에서 데이터를 전처리하는
예시: "I love programming."이라는 문장을 ["I", "love", "programming", "."]으로 분할합니
연관키워드: 토큰화, 자연어 처리, 구문 분석

VectorStore
============================================================
정의: HuggingFace는 자연어 처리를 위한 다양한 사전 훈련된 모델과 도구를 제공하는 라이브러리입니다. 이
예시: HuggingFace의 Transformers 라이브러리를 사용하여 감정 분석, 텍스트 생성 등의 작업을 수행할
```

```
연관키워드: 자연어 처리, 딥러닝, 라이브러리

Digital Transformation
============================================================
huggingface/tokenizers: The current process just got forked, after parallelism h
To disable this warning, you can either:
    - Avoid using `tokenizers` before the fork if possible
    - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(true | fa
```

---

5) `MMR` `Max Marginal Relevance`

- `MMR` (`Maximal Marginal Relevance`) 방식: 쿼리에 대한 관련 항목을 검색할 때 검색된 문서의 `중복` 을 피하는 방법 중 하나
- 단순히 가장 관련성 높은 항목들만을 검색하는 대신, `MMR` 은 쿼리에 대한 `문서의 관련성` 과 이미 선택된 `문서들과의 차별성을 동시에 고려`
  - `search_type` 매개 변수: `mmr` 로 설정 → `MMR` (`Maximal Marginal Relevance`) 검색 알고리즘 사용하기
  - `k`: 반환할 문서 수 *(`기본값 = 4`)*
  - `fetch_k`: `MMR` 알고리즘에 전달할 문서 수 *(`기본값 = 20`)*
  - `lambda_mult`: `MMR` 결과의 다양성 조절 *(`0~1`, `기본값 = 0.5`, `0: 유사도 점수만 고려`, `1: 다양성만 고려`)*

```python
# MMR(Maximal Marginal Relevance) 검색 유형을 지정
retriever = db.as_retriever(
    search_type="mmr",                          # 검색유형 = MMR
    search_kwargs={
        "k": 2,                                 # 반환할 문서 수 = 2
        "fetch_k": 10,                          # 전달할 문서 수 = 10
        "lambda_mult": 0.6                      # MMR 다양성 조절 = 0.6
        }
)

# 관련 문서 검색하기
docs = retriever.invoke("임베딩(Embedding)은 무엇인가요?")

# 관련 문서 출력하기
for doc in docs:
    print(doc.page_content)
    print("============================================================")
```

- 셀 출력

```
정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽
예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.
연관키워드: 데이터 교환, 웹 개발, API

Transformer
============================================================
정의: 토크나이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리에서 데이터를 전처리하는
예시: "I love programming."이라는 문장을 ["I", "love", "programming", "."]으로 분할합니
연관키워드: 토큰화, 자연어 처리, 구문 분석

VectorStore
============================================================
```

---

6) 유사도 점수 임계값 검색 (`similarity_score_threshold`)

- 유사도 점수 임계값을 설정하고 해당 임계값 이상의 점수를 가진 문서만 반환하는 검색 방법을 설정할 수 있음
- 임계값을 적절히 설정 → **관련성이 낮은 문서를 필터링**, 질의와 **가장 유사한 문서만 선별 가능**

- **`search_type`** 매개변수 = **`similarity_score_threshold`**
- **`search_kwargs`** 매개변수: **`{"score_threshold": 0.8}`** 전달 → 유사도 점수 임계값 = 0.8 으로 설정
    - **검색 결과의 유사도 점수가 0.8 이상인 문서만 반한됨을 의미**

```python
# 유사도 점수 임계값으로 설정하기
retriever = db.as_retriever(
    search_type="similarity_score_threshold",           # 검색 유형: "similarity_sc
    search_kwargs={"score_threshold": 0.2},             # 유사도 점수 임계값: 0.2
)

# 출력하기
for doc in retriever.invoke("Word2Vec 은 무엇인가요?"):
    print(doc.page_content)
    print("============================================================")
```

- 셀 출력 (0.2s) (임계값 = 0.2)

```
정의: Word2Vec은 단어를 벡터 공간에 매핑하여 단어 간의 의미적 관계를 나타내는 자연어 처리 기술입니다.
예시: Word2Vec 모델에서 "왕"과 "여왕"은 서로 가까운 위치에 벡터로 표현됩니다.
연관키워드: 자연어 처리, 임베딩, 의미론적 유사성
LLM (Large Language Model)
============================================================
```

- 셀 출력 (임계값 > 0.3)
  - `0.3` 이상으로 설정하니 값이 나오지 않음
  - 교재 내용처럼 `임계값을 0.8` 로 설정했을 경우의 메시지

```
No relevant docs were retrieved using the relevance score threshold 0.8
```

---

7) `top_k 설정`

- 검색 시 사용할 `k` 같은 검색 키워드 인자 (`kwargs`) 지정할 수 있음

- `k` 매개변수: 검색 결과에서 반환할 상위 결과의 개수
  - `search_kwargs` 에서 `k` 매개변수를 `1` 로 설정 → 검색 결과로 반환할 문서의 수를 지정

```
# k 설정
retriever = db.as_retriever(search_kwargs={"k": 1})

# 관련 문서를 검색
docs = retriever.invoke("임베딩(Embedding)은 무엇인가요?")

# 관련 문서를 검색
for doc in docs:
    print(doc.page_content)
    print("============================================================")
```

- 셀 출력

```
정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽
예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.
연관키워드: 데이터 교환, 웹 개발, API

Transformer
============================================================
```

---

8) `동적 설정` (`Configurable`)

- 검색 설정을 동적으로 조정하기 위해 `ConfigurableField` 를 사용

- **`ConfigurableField`** = 검색 매개변수의 `고유 식별자`, `이름`, `설명`을 설정하는 역할
- 검색 설정을 조정하기 위해 **`config`** 매개변수 사용 → 검색 설정 지정
- 검색 설정:
  - **`config`** 매개변수에 전달된 `딕셔너리`의 **`configurable`** `키`에 저장
  - **`검색 쿼리`** 와 함께 전달 → 검색 `쿼리`에 따라 **`동적`으로 조정**

```
from langchain_core.runnables import ConfigurableField

# k 설정
retriever = db.as_retriever(search_kwargs={"k": 1}).configurable_fields(
    search_type=ConfigurableField(
        id="search_type",
        name="Search Type",
        description="The search type to use",
    ),
    search_kwargs=ConfigurableField(
        id="search_kwargs",                          # 검색 매개변수의 고유 식별자를 설정
        name="Search Kwargs",                        # 검색 매개변수의 이름을 설정
        description="The search kwargs to use",      # 검색 매개변수에 대한 설명을 작성
    ),
)
```

- 동적 검색설정을 적용한 예시

```
# 검색 설정을 지정. Faiss 검색에서 k=3로 설정하여 가장 유사한 문서 3개를 반환
config = {"configurable": {"search_kwargs": {"k": 3}}}

# 관련 문서를 검색
docs = retriever.invoke("임베딩(Embedding)은 무엇인가요?", config=config)

# 관련 문서를 검색
for doc in docs:
    print(doc.page_content)
    print("==========================================================")
```

- 셀 출력

```
정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽
예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.
연관키워드: 데이터 교환, 웹 개발, API

Transformer
==========================================================
정의: 토큰은 텍스트를 더 작은 단위로 분할하는 것을 의미합니다. 이는 일반적으로 단어, 문장, 또는 구절일
예시: 문장 "나는 학교에 간다"를 "나는", "학교에", "간다"로 분할합니다.
연관키워드: 토큰화, 자연어 처리, 구문 분석

Tokenizer
==========================================================
```

```
정의: 토크나이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리에서 데이터를 전처리하는
예시: "I love programming."이라는 문장을 ["I", "love", "programming", "."]으로 분할합니
연관키워드: 토큰화, 자연어 처리, 구문 분석

VectorStore
==========================================================
```

```
# 검색 설정을 지정: mmr 검색 설정
config = {
    "configurable": {
        "search_type": "mmr",
        "search_kwargs": {"k": 2, "fetch_k": 10, "lambda_mult": 0.6},
    }
}

# 관련 문서를 검색
docs = retriever.invoke("Word2Vec 은 무엇인가요?", config=config)

# 관련 문서를 검색
for doc in docs:
    print(doc.page_content)
    print("==========================================================")
```

- 셀 출력

```
정의: Word2Vec은 단어를 벡터 공간에 매핑하여 단어 간의 의미적 관계를 나타내는 자연어 처리 기술입니다.
예시: Word2Vec 모델에서 "왕"과 "여왕"은 서로 가까운 위치에 벡터로 표현됩니다.
연관키워드: 자연어 처리, 임베딩, 의미론적 유사성
LLM (Large Language Model)
==========================================================
정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽
예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.
연관키워드: 데이터 교환, 웹 개발, API

Transformer
==========================================================
```

---

9) `Upstage 임베딩과 같이 Query & Passage embedding model이 분리된 경우`

- 기본 retriever는 쿼리와 문서에 대해 동일한 임베딩 모델을 사용
- 하지만 쿼리와 문서에 대해 서로 다른 임베딩 모델을 사용하는 경우가 있음
    - 이러한 경우에는 쿼리 임베딩 모델을 사용하여 쿼리를 임베딩하고, 문서 임베딩 모델을 사용하여 문서를 임베딩함

- 이렇게 하면 쿼리와 문서에 대해 서로 다른 임베딩 모델을 사용할 수 있음

```
from langchain_community.vectorstores import FAISS
from langchain_text_splitters import CharacterTextSplitter
from langchain_community.document_loaders import TextLoader
from langchain_upstage import UpstageEmbeddings

# TextLoader를 사용하여 파일을 로드하기
loader = TextLoader("../10_Retriever/data/appendix-keywords.txt")

# 문서를 로드하기
documents = loader.load()

# 문자 기반으로 텍스트를 분할하는 CharacterTextSplitter를 생성하기
text_splitter = CharacterTextSplitter(
    chunk_size=300,                              # 청크 크기 = 300
    chunk_overlap=0                              # 청크 간 중복 없음
    )

# 로드된 문서 분할하기
split_docs = text_splitter.split_documents(documents)

# Upstage 임베딩 생성하기
doc_embedder = UpstageEmbeddings(model="solar-embedding-1-large-passage")   # 문서

# 분할된 텍스트와 임베딩을 사용하여 FAISS 벡터 데이터베이스를 생성하기
db = FAISS.from_documents(split_docs, doc_embedder)                         # 4.5
```

- 쿼리용 Upstage 임베딩 생성 → 쿼리 문장을 벡터로 변환 → 벡터 유사도 검색 수행하기

```
# 쿼리용 Upstage 임베딩 생성하기
query_embedder = UpstageEmbeddings(model="solar-embedding-1-large-query")   # 쿼리

# 쿼리 문장을 벡터로 변환하기
query_vector = query_embedder.embed_query("임베딩(Embedding)은 무엇인가요?")

# 벡터 유사도 검색을 수행하여 가장 유사한 2개의 문서를 반환하기
db.similarity_search_by_vector(query_vector, k=2)
```

- 셀 출력 (0.6s)

```
[Document(id='e3a78902-0a79-4851-a532-4a8c48d9c2dd', metadata={'source': '../10
Document(id='bbc6b2cc-270a-4f2f-8171-b6b7b1f2f960', metadata={'source': '../10
```

- *next:* ***`문맥 압축 검색기 (ContextualCompressionRetriever)`***
