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

---

5. `Parent Document Retriever`

1) `문서 검색과 문서 분할의 균형 잡기`

- 문서 검색 과정에서 문서를 적절한 크기의 조각(청크)으로 나눌 때 고려해야 할 것
  - **① 작은 문서를 원하는 경우**:
    - 문서의 임베딩이 그 의미를 가장 정확하게 반영할 수 있음
    - 단점: 문서가 너무 길면 임베딩이 의미를 잃어버릴 수 있음
  - **② 각 청크의 맥락이 유지되도록 충분히 긴 문서를 원하는 경우**

- **`ParentDocumentRetriever`의 역할**
  - 위의 두 고려사항 사이의 균형을 맞추기 위해 사용
  - 작동 원리
    - 문서를 작은 조각으로 `나눔` → 조각들을 관리
    - 검색 시: 먼저 이 `작은 조각`들을 찾음 → 조각들이 속한 `원본 문서` (or 더 큰 조각)의 식별자(`ID`)를 통해 `전체적인 맥락` 파악 가능
  - **부모 문서**
    - 작은 조각이 나눠진 **`원본 문서`**: `전체 문서` or `비교적 큰 다른 조각` 일수도 있음
    - **`문서의 의미를 정확하게 파악`** 하면서도 **`전체적인 맥락 유지 가능`**

- **`정리`**
  - **`문서 간의 계층 구조 활용`**: **`ParentDocumentRetriever`** = 문서 검색의 효율성을 높이기 위해 *`문서 간의 계층 구조 활용`*
  - **`검색 성능 향상`**: 관련성 높은 문서를 빠르게 찾아냄 → 주어진 질문에 대한 가장 적합한 답변을 제공하는 문서를 효과적으로 찾아낼 수 있음

---

2) `설정`

- **`TextLoader`** 객체 생성: 여러 개의 텍스트 파일을 로드하기 위함
- **`데이터 로드`**

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

# API 키 정보 로드
load_dotenv()                                   # 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 않음" # API 키 값은 직접 출력하지 않음

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langchain_project:
    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: '{langchain_tracing_v2}').")
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

- 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.
```

```
from langchain.storage import InMemoryStore
from langchain_community.document_loaders import TextLoader
from langchain_chroma import Chroma
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain.retrievers import ParentDocumentRetriever
import warnings
warnings.filterwarnings("ignore")                                    # 5.0s
```

```
# 1단계: Embeddings 사용

embeddings = HuggingFaceEmbeddings(
        model_name="sentence-transformers/all-mpnet-base-v2",              # 768차원
        model_kwargs={'device': 'cpu'},
        encode_kwargs={'normalize_embeddings': True}
    )                                                                      # 6.9s
```

```
# 파일 로드하기
loaders = [
    TextLoader("../10_Retriever/data/appendix-keywords.txt"),
]

docs = []

for loader in loaders:
    # 로더를 사용하여 문서를 로드하고 docs 리스트에 추가하기
    docs.extend(loader.load())
```

---

2) `전체 문서 검색`

- 이 모드에서 `전체 문서` 검색하고자 함
- **`child_splitter`** 만 지정하도록 함 → 나중에 **`parent_spliiter`** 지정해 결과 비교 예정

```
# 자식 분할기 생성하기
child_splitter = RecursiveCharacterTextSplitter(chunk_size=200)

# DB 생성하기
vectorstore = Chroma(
    collection_name="full_documents",
    embedding_function=embeddings
)

store = InMemoryStore()

# Retriever 생성하기
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=store,
    child_splitter=child_splitter,
)                                                                      # 0.2s
```

- **`retriever.add_documents(docs, ids=None)`** 함수 = 문서 목록 추가
  - **`ids`** = **`None`** → 자동으로 생성됨
  - **`add_to_docstore`** = **`False`** → `document` 를 중복으로 추가하지 않음

    - **단, 중복 체크를 위한 `ids` 값이 `필수 값`으로 요구됨**

```
# 문서를 검색기에 추가함

retriever.add_documents(
    docs,                                                    # 문서 목록
    ids=None,                                                # 문서의 고유 식별자 목록
    add_to_docstore=True                                     # 문서 중복 추가하지 않음
    )                                                        # 6.4s
```

- **`두 개의 키 반환`**
    - **`store`** 객체의 **`yield_keys()`** 메서드 호출 → 반환된 키(`key`) 값들을 리스트로 변환하기

```
# 저장소의 모든 키를 리스트로 반환하기

list(store.yield_keys())                                     # ['a7b29e98-1d74-4380-a2fa-c9b75961ea49']
```

- **`벡터 스토어의 검색 기능 호출`**
    - 작은 `chunk`들을 저장하고 있음 → 검색 결과로 작은 청크들이 반환되는 것을 확인할 수 있음
    - **`vectorstore`** 객체의 **`similarity_search`** 메서드 사용 → 유사도 검색 수행

```
# 유사도 검색 수행하기

sub_docs = vectorstore.similarity_search("Word2Vec")
```

- **`sub_docs[0].page_content`** 출력하기

```
# sub_docs 리스트의 첫 번째 요소의 page_content 속성 출력하기

print(sub_docs[0].page_content)
```

- 유사도 검색 수행 (0.1s) → 출력

```
예시: Word2Vec 모델에서 "왕"과 "여왕"은 서로 가까운 위치에 벡터로 표현됩니다.
연관키워드: 자연어 처리, 임베딩, 의미론적 유사성
LLM (Large Language Model)
```

- 전체 `retriever`에서 검색해보기
    - 작은 `chunk`들이 위치한 **`문서를 반환`** → 상대적으로 큰 문서들이 반환될 것

- **`retriever`** 객체의 **`invoke()`** 메서드 사용 → 쿼리와 관련된 문서 검색

```
# 문서를 검색하여 가져오기

retrieved_docs = retriever.invoke("Word2Vec")
```

- 검색된 문서(`retriever_docs[0]`)의 일부 내용 출력하기

```
# 검색된 문서의 문서의 페이지 내용의 길이 출력하기
print(
    f"문서의 길이: {len(retrieved_docs[0].page_content)}",
    end="\n\n====================\n\n",
)

# 문서의 일부 출력하기
print(retrieved_docs[0].page_content[2000:2500])
```

- 셀 출력 (0.1s)

```
문서의 길이: 5733


====================

컴퓨팅을 도입하여 데이터 저장과 처리를 혁신하는 것은 디지털 변환의 예입니다.
연관키워드: 혁신, 기술, 비즈니스 모델
```

```
Crawling

정의: 크롤링은 자동화된 방식으로 웹 페이지를 방문하여 데이터를 수집하는 과정입니다. 이는 검색 엔진 최적화나 데이터 분석에 자주 사용됩니다.
예시: 구글 검색 엔진이 인터넷 상의 웹사이트를 방문하여 콘텐츠를 수집하고 인덱싱하는 것이 크롤링입니다.
연관키워드: 데이터 수집, 웹 스크래핑, 검색 엔진

Word2Vec

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

정의: LLM은 대규모의 텍스트 데이터로 훈련된 큰 규모의 언어 모델을
```

---

3) `더 큰 Chunk 크기 조절`

- **`전체 문서가 너무 커서 있는 그대로 검색하기에 부적합한 경우`**
  - 원시 문서를 **`더 큰 청크로 분할`** → 검색 시에는 `더 큰 청크`로 검색함
  - **`더 작은 청크로 분할`**
  - **`작은 청크들을 인덱싱`**
- 사용 예시
  - **`RecursiveCharacterTextSplitter`** 사용 → `부모 문서`, `자식 문서` 생성
    - 부모 문서: **`chunk_size = 1000`** 으로 설정
    - 자식 문서: **`chunk_size = 200`** 으로 설정 / **`부모 문서보다 작은 크기로 생성됨`**

```
# 부모 문서를 생성하는 데 사용되는 텍스트 분할기
parent_splitter = RecursiveCharacterTextSplitter(chunk_size=1000)

# 자식 문서를 생성하는 데 사용되는 텍스트 분할기
child_splitter = RecursiveCharacterTextSplitter(chunk_size=200)         # 부모보다 작은 문서를 생성해야 함

# 자식 청크를 인덱싱하는 데 사용할 벡터 저장소
vectorstore = Chroma(
    collection_name="split_parents", embedding_function=embeddings
)

# 부모 문서의 저장 계층
store = InMemoryStore()
```

- **`ParentDocumentRetiever`** 초기화 코드
  - **`vectorstore`**: 문서 벡터를 저장하는 벡터 저장소를 지정
  - **`docstore`**: 문서 데이터를 저장하는 문서 저장소를 지정
  - **`child_splitter`**: 하위 문서를 분할하는 데 사용되는 문서 분할기 지정
  - **`parent_spliiter`**: 상위 문서를 분할하는 데 사용되는 분서 분할기 지정

- **`ParentDocumentRetriever`**
  - **계층적 문서 구조 처리** → 상위 문서, 하위 문서를 별도로 분할하고 저장
  - 검색 시 상위 문서와 하위 문서를 효과적으로 활용할 수 있음

```
# Retriever 생성하기
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,                                    # 벡터 저장소 지정
    docstore=store,                                             # 문서 저장소 지정
    child_splitter=child_splitter,                              # 하위 문서 분할기 지정
    parent_splitter=parent_splitter,                            # 상위 문서 분할기 지정
)
```

- **`retriever`** 객체에 **`docs`** 추가하기
  - **`retriever`** 가 검색할 수 있는 문서 집합에 새로운 문서를 추가하는 역할 담당

```
retriever.add_documents(docs)                                    # 문서를 retriever에 추가하기 (6.1s)
```

- **더 큰 청크** 확인해보기

```
# 저장소에서 키를 생성하고 리스트로 변환한 후 길이를 반환하기

len(list(store.yield_keys()))                                    # 7
```

- 기본 벡터 저장소가 여전히 작은 청크를 검색하는지 확인해보기
- **vectorstore** 객체의 **`similarity_search`** 메서드 사용 → 유사도 검색 수행

```
# 유사도 검색 수행하기
sub_docs = vectorstore.similarity_search("Word2Vec")

# sub_docs 리스트의 첫 번째 요소의 page_content 속성을 출력하기
print(sub_docs[0].page_content)
```

- 셀 출력

```
예시: Word2Vec 모델에서 "왕"과 "여왕"은 서로 가까운 위치에 벡터로 표현됩니다.
연관키워드: 자연어 처리, 임베딩, 의미론적 유사성
LLM (Large Language Model)
```

- **retriever** 객체의 **invoke()** 메서드 사용 → 문서 검색하기

```
# 문서를 검색하여 가져오기
retrieved_docs = retriever.invoke("Word2Vec")

# 검색된 문서의 첫 번째 문서의 페이지 내용의 길이를 반환하기
print(retrieved_docs[0].page_content)
```

- 셀 출력 (0.1s)

```
정의: 트랜스포머는 자연어 처리에서 사용되는 딥러닝 모델의 한 유형으로, 주로 번역, 요약, 텍스트 생성 등에 사용됩니다. 이는 Attention 메커니즘을 기
예시: 구글 번역기는 트랜스포머 모델을 사용하여 다양한 언어 간의 번역을 수행합니다.
연관키워드: 딥러닝, 자연어 처리, Attention

HuggingFace

정의: HuggingFace는 자연어 처리를 위한 다양한 사전 훈련된 모델과 도구를 제공하는 라이브러리입니다. 이는 연구자와 개발자들이 쉽게 NLP 작업을 수행
예시: HuggingFace의 Transformers 라이브러리를 사용하여 감정 분석, 텍스트 생성 등의 작업을 수행할 수 있습니다.
연관키워드: 자연어 처리, 딥러닝, 라이브러리

Digital Transformation

정의: 디지털 변환은 기술을 활용하여 기업의 서비스, 문화, 운영을 혁신하는 과정입니다. 이는 비즈니스 모델을 개선하고 디지털 기술을 통해 경쟁력을 높이는
예시: 기업이 클라우드 컴퓨팅을 도입하여 데이터 저장과 처리를 혁신하는 것은 디지털 변환의 예입니다.
연관키워드: 혁신, 기술, 비즈니스 모델

Crawling

정의: 크롤링은 자동화된 방식으로 웹 페이지를 방문하여 데이터를 수집하는 과정입니다. 이는 검색 엔진 최적화나 데이터 분석에 자주 사용됩니다.
예시: 구글 검색 엔진이 인터넷 상의 웹사이트를 방문하여 콘텐츠를 수집하고 인덱싱하는 것이 크롤링입니다.
연관키워드: 데이터 수집, 웹 스크래핑, 검색 엔진

Word2Vec

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

- *next:* ***다중 쿼리 검색기 (MultiQueryRetriever)***
