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

---

4. `RAPTOR` *(Recursive Abstractive Processing for Tree-Organized Retrieval)*

- `RAPTOR` 논문
  - 문서 색인 생성 및 검색에 대한 흥미로운 접근 방식을 제시함
  - RAPTOR 논문
  - `테디노트의 논문 요약글 (노션)`
    - **`leafs`** = **가장** **`low-level`** **의 시작 문서 집합** → 이 문서들은 `임베딩` 되어 `클러스터링` 됨
    - 이후 `클러스터` 는 **`유사한 문서들 간의 정보`** 를 **`더 높은 수준`** (**`더 추상적인`**)으로 **`요약`**
  - **`leafs`** = 다음과 같이 구성될 수 있음
    - **`단일 문서에서의 텍스트 청크`** *(≒ 논문 예시)*
    - **`전체 문서`** *(≒ 아래 예시)*

- `LangChain` 의 **`LCEL`** 문서에 적용하기

- **`환경설정`**

```
# 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.
```

✦

1) 데이터 전처리

- **doc** = **LCEL 문서의 고유한 웹페이지**
- **context** = 2,000토큰 미만 / 10,000토큰 이상까지 다양

- 웹 문서에서 텍스트 데이터 추출 → 텍스트의 토큰 수 계산 → 히스토그램으로 시각화
  - **tiktoken 라이브러리**: 주어진 인코딩 이름에 따라 **문자열의 토큰 수**를 **계산**
  - **RecursiveUrlLoader 클래스**:
    - 지정된 URL에서 웹 문서를 재귀적으로 로드
    - 이 과정에서 BeautifulSoup를 활용 → HTML 문서에서 텍스트를 추출
  - 여러 URL에서 문서를 로드하여 모든 텍스트 데이터 → **하나의 리스트에 모으기**
  - 각 문서 텍스트 → **num_tokens_from_string** 함수 호출 → 토큰 수 계산 → 리스트에 저장
  - **matplotlib**
    - 계산된 토큰 수의 분포를 히스토그램으로 시각화
    - 토큰 수 = x축, 해당 토큰 수를 가진 문서의 빈도수 = y축
  - 히스토그램
    - 데이터의 분포를 이해하는 데 도움
    - 특히 텍스트 데이터의 길이 분포를 시각적으로 파악 가능

```
from langchain_community.document_loaders.recursive_url_loader import RecursiveUrlLoader
from bs4 import BeautifulSoup as Soup
import tiktoken
import matplotlib.pyplot as plt

# 토큰 수 계산
def num_tokens_from_string(string: str, encoding_name: str):
    encoding = tiktoken.get_encoding(encoding_name)
    num_tokens = len(encoding.encode(string))
    return num_tokens                                          # 11.3s
```

- 수정 전 코드 - **LCEL** 문서 로드

```
# LCEL 문서 로드 (수정 전)
url = "https://python.langchain.com/docs/concepts/lcel/"

loader = RecursiveUrlLoader(
    url=url,
    max_depth=20,
    extractor=lambda x: Soup(x, "html.parser").text
)
docs = loader.load()

print(f"로드된 문서 개수: {len(docs)}")
```

- 실행 중 Warning Messages

- 메시지 의미
  - 1번 메시지 의미: **"BeautifulSoup" 라이브러리가 HTML 파서(html.parser)를 사용하여 XML 문서처럼 보이는 것을 파싱하고 있음을 알려줌**

```
XMLParsedAsHTMLWarning: It looks like you're using an HTML parser to parse an XML document.
```

  -
    - 원인: 웹에서 로드되는 파일 중 일부(예: 웹사이트의 sitemap.xml 또는 일부 API 응답)가 HTML이 아닌 XML 형식일 수 있는데, 코드에서 HTML 파서를 사용하도록 지정했기 때문

- 2번 메시지 의미: **XML 파서(lxml 설치 후 features="xml" 사용)를 사용하는 것이 더 안정적일 것이라고 권장**

```
Assuming this really is an XML document, what you're doing might work, but you should know that using an XML p
```

- 원인: `안정적인 파싱`을 위해 `더 적합한 도구`를 `사용`하라는 `제안`

- 실제 메시지

```
/var/folders/h3/l7wnkv352kqftv0t8ctl2ld40000gn/T/ipykernel_94341/3466993954.py:7: XMLParsedAsHTMLWarning: It look

Assuming this really is an XML document, what you're doing might work, but you should know that using an XML pars

If you want or need to use an HTML parser on this document, you can make this warning go away by filtering it. To

    from bs4 import XMLParsedAsHTMLWarning
    import warnings

    warnings.filterwarnings("ignore", category=XMLParsedAsHTMLWarning)

  extractor=lambda x: Soup(x, "html.parser").text
```

```
/Users/jay/.pyenv/versions/lc_env/lib/python3.13/site-packages/langchain_community/document_loaders/recursive_url

Assuming this really is an XML document, what you're doing might work, but you should know that using an XML pars

If you want or need to use an HTML parser on this document, you can make this warning go away by filtering it. To

    from bs4 import XMLParsedAsHTMLWarning
    import warnings

    warnings.filterwarnings("ignore", category=XMLParsedAsHTMLWarning)

  soup = BeautifulSoup(raw_html, "html.parser")
```

- 실행이 30분 넘게 진행 중 → 강제 중단

- 실행이 오래 걸리는 이유: **`RecursiveUrlLoader`** 작동 방식 때문
  - **`재귀적 탐색`**: `RecursiveUrlLoader`는 기본 URL 뿐만 아니라 해당 페이지 내의 `모든 유효한 하위 링크`(`자식 페이지`)를 **`max_depth`** (20으로 설정됨)까지 계속 따라가며 문서를 가져옴
  - **`광범위한 검색`**:
    - `LangChain` 문서 = `매우 방대`하며, `max_depth=20`은 사실상 거의 모든 문서를 긁어오도록 지시하는 것과 같음
    - 이 과정에서 수십, 수백 개의 페이지를 요청하고 `구문 분석`(`Parsing`)해야 하므로 시간이 오래 걸림
  - **`네트워크 지연`**: 각 페이지를 가져올 때마다 네트워크 요청이 발생하며, 이는 로컬 코드 실행보다 훨씬 더 많은 시간을 소모

- `max_depth`, `url_filter` 조정해보기
  - **속도 문제:** `max_depth=20`이 문제의 주원인 → 원하는 문서의 범위에 맞춰 `max_depth`를 **1** or **2** 등으로 줄여보기
  - **경고 문제:**
    - `pip install lxml` 실행 → `BeautifulSoup(x, "lxml").text` 사용 → 경고 없이 더 빠르게 파싱 가능

- 사전에 `VS Code` 터미널에 설치할 것

```
    pip install lxml
```

- `ReculsiveUrlLoader` 문서 로드 시 `TooManyRedirects` 오류 발생

    - 해당 `URL`이 너무 많은 리디렉션을 발생시키기 때문
    - 웹 페이지 주소가 다른 페이지로 계속해서 자동 연결 → 이 연결 횟수가 `ReculsiveUrlLoader`의 내부 제한 (기본 = 30회)을 초과할 경우 = 오류 발생

- **`웹사이트 구조나 설정문제 or 해당 페이지가 더이상 유효하지 않아 계속 다른 곳으로 보내지는 경우에 발생할 수 있음`**
    - *해당 페이지 유효함*

- **`오류 해결 방법`**
    - **`max_depth`** 줄이기 → `✓`
        - 교재: `max_depth = 20` → `max_depth = 5`
    - **`url_filter`** 사용
        - 특정 패턴의 `URL`만 포함하거나 제외하도록 매개변수 사용 → 문제가 되는 리디렉션 체인이 시작되는 `URL` 건너뛸 수 있음
        - **`아래에서 시도해보기`**
    - **`문제가 되는 URL 식별 및 제외`**
        - *`ex: https://python.langchain.com/docs/integrations/providers/openai/`*
    - **`다른 로더 사용`**: **`WebBaseLoader`** 등 단일 페이지를 만드는 데 더 적합한 로더를 사용할 수도 있음
    - **`네트워크 or 웹사이트 문제 확인`**

```
from langchain_community.document_loaders import RecursiveUrlLoader
from bs4 import BeautifulSoup
import time
import warnings
from bs4 import XMLParsedAsHTMLWarning
import logging

# 경고 메시지 필터링: XMLParsedAsHTMLWarning을 무시하여 콘솔을 깔끔하게 유지
warnings.filterwarnings("ignore", category=XMLParsedAsHTMLWarning)

# 로깅 설정 (선택 사항: 로더가 어떤 URL을 탐색하는지 확인하려면 주석 해제)
# logging.basicConfig(level=logging.INFO)

print("--- LangChain LCEL 문서 로드 시작 ---")
start_time = time.time()

# 1. 문서 로드 URL 정의
url = "https://python.langchain.com/docs/expression_language/"

# 2. RecursiveUrlLoader 설정
#    - max_depth=2: 로딩 시간을 크게 단축하기 위해 재귀 깊이를 5로 제한
#      (기본 URL + 그 페이지에서 직접 연결된 하위 페이지까지만 탐색)
#    - extractor: lxml 파서를 사용하여 파싱 경고를 제거하고 성능을 개선
loader = RecursiveUrlLoader(
    url=url,
    max_depth=2,                                          # 탐색 깊이를 제한하여 속도 개선
    extractor=lambda x: BeautifulSoup(x, "lxml").text   # 'lxml' 파서를 사용하여 안정성 및 속도 개선
)

try:
    # 3. 문서 로드 실행
    # 이 과정은 max_depth=2로 인해 이전보다 훨씬 빠르게 완료될 것
    docs = loader.load()

    end_time = time.time()
    elapsed_time = end_time - start_time

    print("-" * 40)
    print("✅ 문서 로드 성공 및 분석 완료!")
    print(f"총 로드된 문서 개수: {len(docs)}")
    print(f"소요 시간: {elapsed_time:.2f}초")
    print("-" * 40)

    # 로드된 문서의 첫 번째 문서 일부 출력
    if docs:
        print("\n[첫 번째 문서 미리보기]")
        print(f"소스: {docs[0].metadata.get('source', '알 수 없음')}")
        print(f"내용 (첫 200자): {docs[0].page_content[:200]}...")
except Exception as e:
    print(f"\n❌ 문서 로드 중 예상치 못한 오류 발생: {e}")
```

- **`max_depth = 2`** → ⭕ (`1m 49.9s`)

```
--- LangChain LCEL 문서 로드 시작 ---
----------------------------------------
✅ 문서 로드 성공 및 분석 완료!
총 로드된 문서 개수: 229
소요 시간: 109.90초
----------------------------------------

[첫 번째 문서 미리보기]
소스: https://python.langchain.com/docs/expression_language/
내용 (첫 200자):



LangChain Expression Language (LCEL) | ü¶úÔ¶èüîó LangChain









Skip to main contentThese docs will be deprecated and no longer maintained with the release of LangChain v1.0 in
```

```
# PydanticOutputParser를 사용한 LCEL 문서 로드 (기본 LCEL 문서 외부)
url = "https://python.langchain.com/docs/modules/model_io/output_parsers/quick_start"

loader = RecursiveUrlLoader(
    url=url,
    max_depth=1,
    extractor=lambda x: Soup(x, "html.parser").text
)
docs_pydantic = loader.load()
print(f"로드된 문서 개수: {len(docs_pydantic)}")
```

- 로드된 문서 개수: 1 (`0.3s`)

```
# Self Query를 사용한 LCEL 문서 로드 (기본 LCEL 문서 외부)
url = "https://python.langchain.com/docs/modules/data_connection/retrievers/self_query/"

loader = RecursiveUrlLoader(
    url=url,
    max_depth=1,
    extractor=lambda x: Soup(x, "html.parser").text
)
docs_sq = loader.load()
print(f"로드된 문서 개수: {len(docs_sq)}")
```

- 로드된 문서 개수: 1 (`0.5s`)

```
# 문서 텍스트
docs.extend([*docs_pydantic, *docs_sq])
docs_texts = [d.page_content for d in docs]
```

```
print(type(docs_texts))
print(len(docs_texts))
```

- <class 'list'>
- 231

```
# 각 문서에 대한 토큰 수 계산
counts = [num_tokens_from_string(d, "cl100k_base") for d in docs_texts]

print(type(counts))
print(len(counts))
```

- <class 'list'>
- 231

- **토큰 수 계산 및 히스토그램**

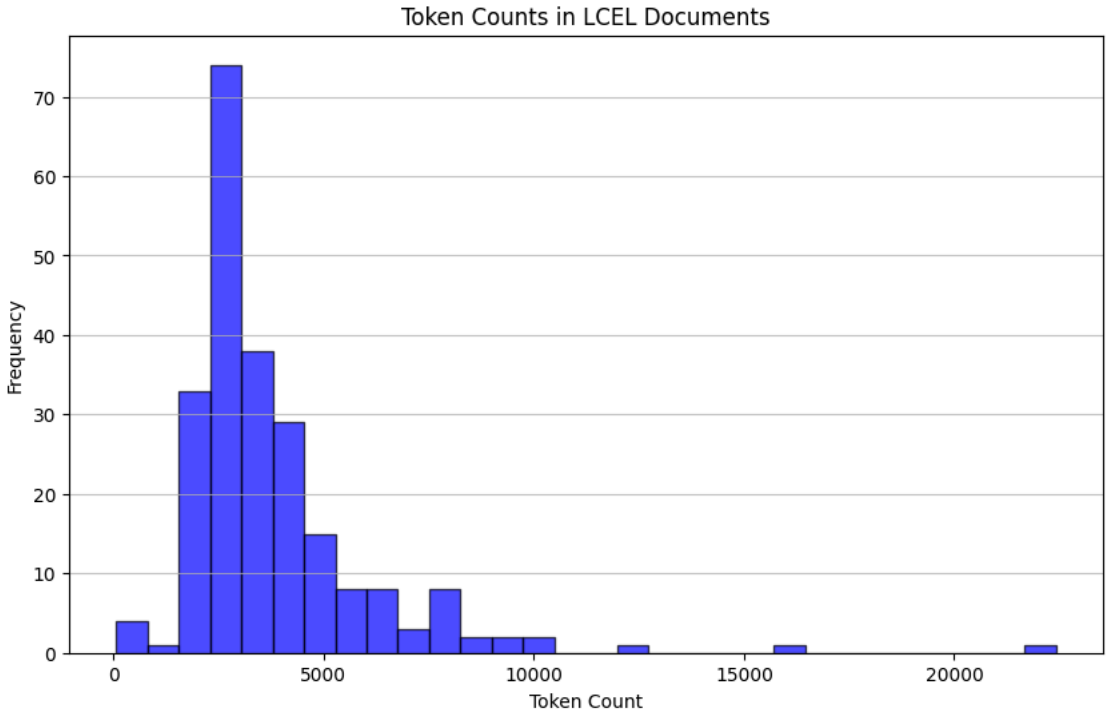
```
# 토큰 수의 히스토그램 그리기
plt.figure(figsize=(10, 6))
plt.hist(counts, bins=30, color="blue", edgecolor="black", alpha=0.7)
plt.title("Token Counts in LCEL Documents")
plt.xlabel("Token Count")
plt.ylabel("Frequency")
plt.grid(axis="y", alpha=0.75)

# 히스토그램 표시
plt.show
```

- 문서 텍스트 정렬, 연결 → 토큰 수를 계산하는 과정 설명하기
  - 문서 (`docs`)를 **메타데이터** **`"source"`** 키를 기준으로 정렬
  - 정렬된 문서 리스트를 **역순** 으로 뒤집기
  - 역순으로 된 문서의 내용을 **특정 구분자** (**`"\n\n\n --- \n\n\n"`**)를 사용 → 연결
  - **`num_tokens_from_string`** 함수
    - 연결된 내용의 토큰 수 계산 → 출력
    - 이때, **`"cl100k_base"`** 모델 사용

```
# 문서 텍스트 연결하기
# 문서를 출처 메타데이터 기준으로 정렬하기
d_sorted = sorted(docs, key=lambda x: x.metadata["source"])

d_reversed = list(reversed(d_sorted))               # 정렬된 문서를 역순으로 배열

concatenated_content = "\n\n\n --- \n\n\n".join(
    [
        # 역순으로 배열된 문서의 내용을 연결하기
        doc.page_content
        for doc in d_reversed
    ]
)

print(
    "Num tokens in all context: %s"                 # 모든 문맥에서의 토큰 수 출력
    % num_tokens_from_string(concatenated_content, "cl100k_base")
)
```

- **`Num tokens in all context: 877518`** (`0.4s`)

- 히스토그램 (`0.3s`) + 토큰 수 계산 과정 설명



  - 토큰 수 계산 과정 설명: `Num tokens in all context: 877518`

- **`RecursiveCharacterTextSplitter`** → 텍스트를 분할하는 과정 설명하기
  - **`chunk_size_tok`** 변수 설정 → 각 텍스트 청크의 크기를 `2000` 토큰으로 지정
  - **`RecursiveCharacterTextSplitter`** **`from_tiktoken_encoder`** 메소드 사용 → 텍스트 분할기 초기화
    - **청크 크기 (`chunk_size`) = `chunk_size_tok`**
    - **청크 간 겹침 (`chunk_overlap`) = `0`**
  - 초기화된 텍스트 분할기의 **`split_text`** 메소드 호출
    - **`concatenated_content`** 변수에 저장된 연결된 텍스트를 분할
    - 분할 결과 = **`texts_split`** **변수에 저장**

```
# 텍스트 분할을 위한 코드
from langchain_text_splitters import RecursiveCharacterTextSplitter

chunk_size_tok = 2000                                # 토큰의 청크 크기 설정

# 재귀적 문자 텍스트 분할기 초기화
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=chunk_size_tok,                       # 청크 크기 설정
    chunk_overlap=0                                  # 청크 중복 허용하지 않음
)

# 주어진 텍스트 분할하기
texts_split = text_splitter.split_text(
    concatenated_content
)                                                    # 1.9s
```

```
print(type(text_splitter))      # <class 'langchain_text_splitters.character.RecursiveCharacterTextSplitter'>
```

```
print(type(texts_split))
print(len(texts_split))
```

- <class 'list'>
- 724

---

2) `모델`

- 다양한 모델 테스트 가능
- 교재에서는 `Claude` + `OpenAIEmbeddings`으로 챗봇 모델 구현
  - `OpenAIEmbeddings` 인스턴스화 → `OpenAI`의 임베딩 기능 초기화
  - `LLM` 모델의 `temperature` = `0`으로 설정 → 챗봇 모델 초기화

- **`Cache Embedding`** 사용하기

```
from langchain_huggingface import HuggingFaceEmbeddings
import warnings

# 경고 무시
warnings.filterwarnings("ignore")

# 임베딩 모델 생성하기
# HuggingFace Embeddings 사용
embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
)

print("✅ hugging-face 임베딩 모델 로딩 완료!")
```

- ✅ hugging-face 임베딩 모델 로딩 완료! (`9.3s`)

```
from langchain_huggingface import HuggingFaceEmbeddings
from langchain.embeddings import CacheBackedEmbeddings
from langchain.storage import LocalFileStore

# 경고 무시
warnings.filterwarnings("ignore")
```

```
# 캐시가 저장될 디렉토리 설정: 절대 경로로 명확히 지정
cache_dir = "/Users/jay/Projects/20250727-langchain-note/12_RAG/cache/"
store = LocalFileStore(cache_dir)
print(f"캐시 저장소 경로 설정 완료: {cache_dir}")

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

cached_embeddings = CacheBackedEmbeddings.from_bytes_store(
    embd, store, namespace=embd.model_name
)

print("✅ Cache Embedding 인스턴스 생성 완료")
```

- 캐시 저장소 경로 설정 완료: `/Users/jay/Projects/20250727-langchain-note/12_RAG/cache/`
- ✅ Cache Embedding 인스턴스 생성 완료 (`2.8s`)

- **`요약 LLM 초기화하기`**

- 재귀적 요약 = `LLM` 반복적 호출 → `API` 할당량 초과 → 코드의 무한정 루프에 빠짐

- **`HuggingFace Pipeline`** 사용하기 (*`local model`*)
  - `BART` 기반 모델: 최대 입력 길이가 비교적 *짧은 편* (`보통 1024 토큰`) → 긴 클러스터 요약에 부적합

```
from langchain_community.llms.huggingface_pipeline import HuggingFacePipeline
from transformers import pipeline

# LLM 대신 요약 파이프라인 설정 (작업 부하가 CPU/로컬로 전환됨)

summarization_pipeline = pipeline(
    "summarization",
    model="facebook/bart-large-cnn",
    device=-1,              # CPU 사용 (-1) 또는 GPU 사용 (0)
)

llm = HuggingFacePipeline(pipeline=summarization_pipeline)
```

*

- `T5` 모델: *긴 입력 시퀀스 처리에 더 유연*

```
from langchain_community.llms.huggingface_pipeline import HuggingFacePipeline
from transformers import pipeline

# LLM 대신 요약 파이프라인 설정 (작업 부하가 CPU/로컬로 전환됨)

summarization_pipeline = pipeline(
    "summarization",
    model="t5-base",        # 모델 = T5
    device=-1,              # CPU 사용 (-1) 또는 GPU 사용 (0)
)

llm = HuggingFacePipeline(pipeline=summarization_pipeline)
```

```
import getpass
import os

GOOGLE_API_KEY=os.environ.get("GOOGLE_API_KEY2")

if not os.environ.get("GOOGLE_API_KEY"):
```

```
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter API key for Google Gemini: ")

from langchain.chat_models import init_chat_model

model = init_chat_model(
    "gemini-2.5-flash",
    model_provider="google_genai"
    )

print("✅ second 계정: gemini-2.5-flash 초기화")
```

- ✅ second 계정: gemini-2.5-flash 초기화

```
E0000 00:00:1759934960.929177 1301213 alts_credentials.cc:93] ALTS creds ignored. Not running on GCP and untruste
```

---

3) `트리 구축`

- ① **`GMM`** *(가우시안 혼합 모델)*
  - 다양한 `클러스터`에 걸쳐 `데이터 포인트`의 `분포` → `모델링`
  - 모델의 `베이지안 정보 기준`(`BIC`)을 `평가` → **`최적의 클러스터 수`** **결정**
- ② **`UMAP`** *(Uniform Manifold Approximation and Projection)*
  - **`클러스터링 지원`**
  - **`고차원 데이터`의 `차원 축소`**
  - **`데이터 포인트의 유사성에 기반`** → **자연스러운 그룹화를 강조하는 데 도움을 줌**
- ③ **`지역 및 전역 클러스터링`**
  - **`다양한 규모`**에서 `데이터`를 `분석`하는 데 사용
  - 데이터 내의 `세밀한 패턴`과 `더 넓은 패턴` 모두를 효과적으로 포착
- ④ **`임계값 설정`**
  - `GMM`의 맥락에서 클러스터 멤버십을 결정하기 위해 적용됨
  - `확률 분포`를 기반으로 함: `데이터 포인트`를 `≥` `1` **클러스터에 할당**

- `GMM` 및 `임계값 설정`에 대한 코드 출처: (Sarthi et al)
  - 원본 저장소
  - 소소한 조정
- **`global_cluster_embeddings`** 함수 → **`UMAP`** **사용** *(임베딩의 글로벌 차원 축소 수행 목적)*
  - **a. `UMAP` 사용**: 입력된 `임베딩`(`embeddings`) → `지정된 차원`(`dim`)으로 **`차원 축소`**
  - **b. `n_neighbors`**
    - 각 포인트를 고려할 **`이웃의 수`** **지정**
    - 제공되지 않을 경우 = 기본 설정 = **`임베딩 수의 제곱근`**
  - **c. `metric`**: `UMAP`에 사용될 **`거리 측정 기준`을 지정**
  - **d. `결과`**: 지정된 차원으로 `축소`된 임베딩이 **`numpy 배열`로 반환**
- 사전에 `VS Code` 터미널에 설치할 것

```
    pip install umap-learn
```

```
from typing import Dict, List, Optional, Tuple

import numpy as np
```

```
import pandas as pd
import umap
from langchain.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from sklearn.mixture import GaussianMixture

RANDOM_SEED = 42                                    # 재현성을 위한 고정된 시드 값

### --- 위의 인용된 코드에서 주석과 문서화를 추가함 --- ###

def global_cluster_embeddings(
    embeddings: np.ndarray,
    dim: int,
    n_neighbors: Optional[int] = None,
    metric: str = "cosine",
) -> np.ndarray:

    """
    UMAP을 사용하여 임베딩의 전역 차원 축소를 수행합니다.

    매개변수:
    - embeddings: numpy 배열로 된 입력 임베딩.
    - dim: 축소된 공간의 목표 차원.
    - n_neighbors: 선택 사항; 각 점을 고려할 이웃의 수.
                   제공되지 않으면 임베딩 수의 제곱근으로 기본 설정됩니다.
    - metric: UMAP에 사용할 거리 측정 기준.

    반환값:
    - 지정된 차원으로 축소된 임베딩의 numpy 배열.
    """

    if n_neighbors is None:
        n_neighbors = int((len(embeddings) - 1) ** 0.5)

    return umap.UMAP(
        n_neighbors=n_neighbors, n_components=dim, metric=metric
    ).fit_transform(embeddings)                          # 4.0s
```

- `local_cluster_embeddings()`: 임베딩 데이터에 대해 지역 차원 축소를 수행하는 함수 구현하기
  - 입력된 임베딩 (`embeddings`) → `UMAP` → 지정된 차원 (`dim`)으로 차원 축소
  - 차원 축소 과정: 각 점에 대해 고려할 이웃의 수(`num_neighbors`)와 거리 측정 메트릭(`metric`)을 파라미터로 사용
  - 최종 반환: **차원이 축소된 임베딩을 numpy 배열**

```
def local_cluster_embeddings(
    embeddings: np.ndarray, dim: int, num_neighbors: int = 10, metric: str = "cosine"
) -> np.ndarray:
    """
    임베딩에 대해 지역 차원 축소를 수행합니다. 이는 일반적으로 전역 클러스터링 이후에 사용됩니다.

    매개변수:
    - embeddings: numpy 배열로서의 입력 임베딩.
    - dim: 축소된 공간의 목표 차원 수.
    - num_neighbors: 각 점에 대해 고려할 이웃의 수.
    - metric: UMAP에 사용할 거리 측정 기준.

    반환값:
    - 지정된 차원으로 축소된 임베딩의 numpy 배열.
    """
    return umap.UMAP(
        n_neighbors=num_neighbors, n_components=dim, metric=metric
    ).fit_transform(embeddings)
```

- `get_optimal_clusters()`
  - 주어진 임베딩 데이터를 기반으로 최적의 클러스터 수를 결정하는 데 사용됨
  - **가우시안 혼합 모델** (**`Gaussian Mixture Model`**) **사용 → 베이지안 정보 기준** (**`Bayesian Information Criterion`**, **`BIC`**) **계산 → 수행됨**

- 매개변수 설명
  - **입력 임베딩** (`embeddings`) = **numpy 배열**
  - **최대 클러스터 수**
    - `max_clusters` = 고려할 클러스터의 최대 수
    - 기본값 = 50
  - **`random_state`** = **`42`**
    - 재현성을 위한 난수 상태 = 고정된 값 사용

- 함수 → 입력 임베딩에 대해 여러 클러스터 수를 시도 → 각각에 대한 BIC 값을 계산
- **최소 BIC 값을 가지는 클러스터 수** = **최적의 클러스터 수**로 결정 → 반환

```
def get_optimal_clusters(
    embeddings: np.ndarray, max_clusters: int = 50, random_state: int = RANDOM_SEED
) -> int:
    """
    가우시안 혼합 모델(Gaussian Mixture Model)을 사용하여 베이지안 정보 기준(BIC)을 통해 최적의 클러스터 수를 결정합니다.

    매개변수:
    - embeddings: numpy 배열로서의 입력 임베딩.
    - max_clusters: 고려할 최대 클러스터 수.
    - random_state: 재현성을 위한 시드.

    반환값:
    - 발견된 최적의 클러스터 수를 나타내는 정수.
    """

    # 최대 클러스터 수와 임베딩의 길이 중 작은 값을 최대 클러스터 수로 설정
    max_clusters = min(
        max_clusters, len(embeddings)
    )

    # 1부터 최대 클러스터 수까지의 범위를 생성
    n_clusters = np.arange(1, max_clusters)

    # BIC 점수를 저장할 리스트
    bics = []

    # 각 클러스터 수에 대해 반복
    for n in n_clusters:
        # 가우시안 혼합 모델 초기화
        gm = GaussianMixture(
            n_components=n, random_state=random_state, reg_covar=1e-4       # 안정화 매개변수 추가
        )
        gm.fit(embeddings)                      # 임베딩에 대해 모델 학습
        bics.append(gm.bic(embeddings))         # 학습된 모델의 BIC 점수를 리스트에 추가
    return n_clusters[np.argmin(bics)]          # BIC 점수가 가장 낮은 클러스터 수를 반환
```

- **`GMM_cluster()`** 함수
  - 임베딩 → **가우시안 혼합 모델** (`GMM`) 사용 → **클러스터링**
  - **확률 임계값** 기반

- 매개변수 설명
  - **입력된 임베딩** (`embeddings`) = **numpy 배열**
  - **`threshold`** = 임베딩을 특정 클러스터에 할당하기 위한 확률 임계값
  - `random_state` = 결과의 재현성을 위한 시드 값
  - **`get_optimal_clusters` 함수** 호출 = **최적의 클러스터 수**를 **결정하기 위함**
  - 결정된 클러스터 수를 바탕으로 가우시안 혼합 모델을 초기화 → 입력된 임베딩에 대해 학습 수행함
  - 각 임베딩에 대한 클러스터 할당 확률 계산 → 이 확률이 주어진 임계값을 초과하는 경우 해당 임베딩을 클러스터에 할당
  - 최종 반환: **임베딩의 클러스터 레이블**과 **결정된 클러스터 수**를 **튜플**

```
def GMM_cluster(embeddings: np.ndarray, threshold: float, random_state: int = 0):
    """
    확률 임계값을 기반으로 가우시안 혼합 모델(GMM)을 사용하여 임베딩을 클러스터링합니다.

    매개변수:
    - embeddings: numpy 배열로서의 입력 임베딩.
    - threshold: 임베딩을 클러스터에 할당하기 위한 확률 임계값.
    - random_state: 재현성을 위한 시드.

    반환값:
    - 클러스터 레이블과 결정된 클러스터 수를 포함하는 튜플.
    """

    # 최적의 클러스터 수 구하기
    n_clusters = get_optimal_clusters(embeddings)

    # 가우시안 혼합 모델 초기화하기
    gm = GaussianMixture(n_components=n_clusters, random_state=random_state)

    # 임베딩에 대해 모델 학습하기
    gm.fit(embeddings)
```

```
    # 임베딩이 각 클러스터에 속할 확률 예측하기
    probs = gm.predict_proba(
        embeddings
    )

    # 임계값을 초과하는 확률을 가진 클러스터를 레이블로 선택하기
    labels = [np.where(prob > threshold)[0] for prob in probs]
    return labels, n_clusters                          # 레이블과 클러스터 수를 튜플로 반환
```

- **`perform_clustering()`**
  - 임베딩에 대해 차원 축소, 가우시안 혼합 모델을 사용한 글로벌 클러스터링, 그리고 각 글로벌 클러스터 내에서의 로컬 클러스터링을 수행하여 클러스터링 결과를 반환

- 매개변수 설명
  - **`차원 축소`**:
    - 입력된 임베딩(embeddings) → 차원 축소 수행
    - UMAP 사용 → 지정된 차원(dim)으로 임베딩의 차원을 축소하는 과정 포함
  - **`글로벌 클러스터링 수행`**
    - 차원이 축소된 임베딩 → 가우시안 혼합 모델(GMM) 사용
    - 클러스터 할당은 `주어진 확률 임계값`(`threshold`)을 기준 → 결정
  - **각 글로벌 클러스터 내 `추가적인 로컬 클러스터링 수행`**
    - 글로벌 클러스터링 결과 → 각 글로벌 클러스터에 속한 임베딩들만을 대상으로 **`다시 차원 축소 및 GMM 클러스터링 진행`**
  - 최종 반환
    - `모든 임베딩`에 대해 글로벌 및 로컬 `클러스터 ID`를 할당
    - 각 임베딩이 속한 클러스터 `ID를 담은 리스트`를 반환 = 임베딩의 순서에 따라 각 임베딩에 대한 `클러스터 ID 배열`을 포함

- 고차원 데이터의 클러스터링을 위해 글로벌 및 로컬 차원에서의 클러스터링을 결합한 접근 방식을 제공 → 더 세분화된 클러스터링 결과를 얻을 수 있으며, 복잡한 데이터 구조를 보다 효과적으로 분석 가능

```
def perform_clustering(
    embeddings: np.ndarray,
    dim: int,
    threshold: float,
) -> List[np.ndarray]:
    """
    임베딩에 대해 차원 축소, 가우시안 혼합 모델을 사용한 클러스터링, 각 글로벌 클러스터 내에서의 로컬 클러스터링을 순서대로 수행합니다.

    매개변수:
    - embeddings: numpy 배열로 된 입력 임베딩입니다.
    - dim: UMAP 축소를 위한 목표 차원입니다.
    - threshold: GMM에서 임베딩을 클러스터에 할당하기 위한 확률 임계값입니다.

    반환값:
    - 각 임베딩의 클러스터 ID를 포함하는 numpy 배열의 리스트입니다.
    """
    if len(embeddings) <= dim + 1:
        # 데이터가 충분하지 않을 때 클러스터링 피하기
        return [np.array([0]) for _ in range(len(embeddings))]

    # 글로벌 차원 축소
    reduced_embeddings_global = global_cluster_embeddings(embeddings, dim)
    # 글로벌 클러스터링
    global_clusters, n_global_clusters = GMM_cluster(
        reduced_embeddings_global, threshold
    )

    all_local_clusters = [np.array([]) for _ in range(len(embeddings))]
    total_clusters = 0

    # 각 글로벌 클러스터를 순회하며 로컬 클러스터링 수행
    for i in range(n_global_clusters):
        # 현재 글로벌 클러스터에 속하는 임베딩 추출
        global_cluster_embeddings_ = embeddings[
            np.array([i in gc for gc in global_clusters])
        ]

        if len(global_cluster_embeddings_) == 0:
            continue
        if len(global_cluster_embeddings_) <= dim + 1:
            # 작은 클러스터는 직접 할당으로 처리
            local_clusters = [np.array([0]) for _ in global_cluster_embeddings_]
```

```
            n_local_clusters = 1
        else:
            # 로컬 차원 축소 및 클러스터링
            reduced_embeddings_local = local_cluster_embeddings(
                global_cluster_embeddings_, dim
            )
            local_clusters, n_local_clusters = GMM_cluster(
                reduced_embeddings_local, threshold
            )

        # 로컬 클러스터 ID 할당, 이미 처리된 총 클러스터 수를 조정
        for j in range(n_local_clusters):
            local_cluster_embeddings_ = global_cluster_embeddings_[
                np.array([j in lc for lc in local_clusters])
            ]
            indices = np.where(
                (embeddings == local_cluster_embeddings_[:, None]).all(-1)
            )[1]
            for idx in indices:
                all_local_clusters[idx] = np.append(
                    all_local_clusters[idx], j + total_clusters
                )

        total_clusters += n_local_clusters

    return all_local_clusters
```

- **`embed()`**: 텍스트 문서의 목록에 대한 임베딩을 생성하는 함수 구현하기
  - 입력= `텍스트 문서`의 `목록`(`texts`)
  - `embd 객체`의 `embed_documents` → 텍스트 문서의 임베딩 생성
  - **`numpy.ndarray` 형태 → 변환 → 반환**

```
def perform_clustering(
    embeddings: np.ndarray,
    dim: int,
    threshold: float,
) -> List[np.ndarray]:

    """
    임베딩에 대해 차원 축소, 가우시안 혼합 모델을 사용한 클러스터링, 각 글로벌 클러스터 내에서의 로컬 클러스터링을 순서대로 수행합니다.

    매개변수:
    - embeddings: numpy 배열로 된 입력 임베딩입니다.
    - dim: UMAP 축소를 위한 목표 차원입니다.
    - threshold: GMM에서 임베딩을 클러스터에 할당하기 위한 확률 임계값입니다.

    반환값:
    - 각 임베딩의 클러스터 ID를 포함하는 numpy 배열의 리스트입니다.
    """

    if len(embeddings) <= dim + 1:
        # 데이터가 충분하지 않을 때 클러스터링 피하기
        return [np.array([0]) for _ in range(len(embeddings))]

    # 글로벌 차원 축소
    reduced_embeddings_global = global_cluster_embeddings(embeddings, dim)

    # 글로벌 클러스터링
    global_clusters, n_global_clusters = GMM_cluster(
        reduced_embeddings_global, threshold
    )

    all_local_clusters = [np.array([]) for _ in range(len(embeddings))]

    total_clusters = 0

    # 각 글로벌 클러스터를 순회하며 로컬 클러스터링 수행
    for i in range(n_global_clusters):
        # 현재 글로벌 클러스터에 속하는 임베딩 추출
        global_cluster_embeddings_ = embeddings[
            np.array([i in gc for gc in global_clusters])
        ]

        if len(global_cluster_embeddings_) == 0:
            continue
        if len(global_cluster_embeddings_) <= dim + 1:
            # 작은 클러스터는 직접 할당으로 처리
            local_clusters = [np.array([0]) for _ in global_cluster_embeddings_]
            n_local_clusters = 1
        else:
            # 로컬 차원 축소 및 클러스터링
```

```
            reduced_embeddings_local = local_cluster_embeddings(
                global_cluster_embeddings_, dim
            )
            local_clusters, n_local_clusters = GMM_cluster(
                reduced_embeddings_local, threshold
            )

        # 로컬 클러스터 ID 할당, 이미 처리된 총 클러스터 수를 조정
        for j in range(n_local_clusters):
            local_cluster_embeddings_ = global_cluster_embeddings_[
                np.array([j in lc for lc in local_clusters])
            ]
            indices = np.where(
                (embeddings == local_cluster_embeddings_[:, None]).all(-1)
            )[1]
            for idx in indices:
                all_local_clusters[idx] = np.append(
                    all_local_clusters[idx], j + total_clusters
                )

        total_clusters += n_local_clusters

    return all_local_clusters
```

- 텍스트 문서의 목록에 대한 임베딩을 생성하는 함수 **`embed`** **구현하기**
  - 입력 = **`텍스트 문서의 목록`** (`texts`)
  - **`embd`** **객체의** **`embed_documents`** **메소드 → 텍스트 문서의** **`임베딩 생성`**
  - 생성된 임베딩 = **`numpy.ndarray`** **형태로** **`변환`** **하여** **`반환`**

```
def embed(texts):

    # 텍스트 문서 목록에 대한 임베딩 생성하기
    #
    # 이 함수는 `embd` 객체가 존재한다고 가정하며, 이 객체는 텍스트 목록을 받아 그 임베딩을 반환하는 `embed_documents` 메소드를 가지고 있음
    #
    # 매개변수:
    # - texts: List[str], 임베딩할 텍스트 문서의 목록
    #
    # 반환값:
    # - numpy.ndarray: 주어진 텍스트 문서들에 대한 임베딩 배열
    text_embeddings = embd.embed_documents(
        texts
    )  # 텍스트 문서들의 임베딩 생성하기

    # 임베딩을 numpy 배열로 변환하기
    text_embeddings_np = np.array(text_embeddings)

    # 임베딩된 numpy 배열을 반환함
    return text_embeddings_np
```

- **`embed_cluster_texts()`**
  - `텍스트 목록`을 `임베딩`하고 `클러스터링` → 원본 텍스트, 해당 임베딩, 그리고 `할당된 클러스터 라벨`을 `포함`하는 **`pandas.DataFrame`** **을 반환**

- 매개변수 설명
  - 주어진 텍스트 목록 → `임베딩 생성`
  - **`perform_clustering()`**: 생성된 임베딩 기반 → `클러스터링 수행`
  - **`pandas.DataFrame`** **초기화**: 결과 저장 목적
  - **`DataFrame`** = `원본 텍스트`, `임베딩 리스트`, `클러스터 라벨`을 각각 저장함

- 이 함수는 텍스트 데이터의 임베딩 생성과 클러스터링을 하나의 단계로 결합하여, 텍스트 데이터의 구조적 분석과 그룹화를 용이하게 함

```
def embed_cluster_texts(texts):

    """
    텍스트 목록을 임베딩하고 클러스터링하여, 텍스트, 그들의 임베딩, 그리고 클러스터 라벨이 포함된 DataFrame을 반환합니다.

    이 함수는 임베딩 생성과 클러스터링을 단일 단계로 결합합니다. 임베딩에 대해 클러스터링을 수행하는 `perform_clustering` 함수의 사전 정의된 존재를 가정합니다.

    매개변수:
    - texts: List[str], 처리될 텍스트 문서의 목록입니다.

    반환값:
```

```
    - pandas.DataFrame: 원본 텍스트, 그들의 임베딩, 그리고 할당된 클러스터 라벨이 포함된 DataFrame입니다.
    """

    # 임베딩 생성
    text_embeddings_np = embed(texts)

    # 임베딩에 대해 클러스터링 수행
    cluster_labels = perform_clustering(
        text_embeddings_np, 10, 0.1
    )

    # 결과를 저장할 DataFrame 초기화
    df = pd.DataFrame()

    # 원본 텍스트 저장
    df["text"] = texts

    # DataFrame에 리스트로 임베딩 저장
    df["embd"] = list(text_embeddings_np)

    # 클러스터 라벨 저장
    df["cluster"] = cluster_labels

    return df
```

- **`fmt_txt()`** : `pandas`의 `DataFrame`에서 `텍스트 문서`를 `단일 문자열`로 `포맷팅`
  - 입력 파라미터로 `DataFrame`을 받으며, 이 `DataFrame`은 포맷팅할 텍스트 문서를 포함한 `'text'` 컬럼을 가져야 함
  - 모든 텍스트 문서 = `특정 구분자`(`"--- --- \n --- ---"`) 사용 + 연결 → `단일 문자열`로 반환
  - 함수: 연결된 텍스트 문서를 포함하는 `단일 문자열`을 반환

```
def fmt_txt(df: pd.DataFrame) -> str:
    """
    DataFrame에 있는 텍스트 문서를 단일 문자열로 포맷합니다.

    매개변수:
    - df: 'text' 열에 포맷할 텍스트 문서가 포함된 DataFrame.

    반환값:
    - 모든 텍스트 문서가 특정 구분자로 결합된 단일 문자열.
    """

    # 'text' 열의 모든 텍스트를 리스트로 변환
    unique_txt = df["text"].tolist()

    # 텍스트 문서들을 특정 구분자로 결합하여 반환
    return "--- --- \n --- --- ".join(
        unique_txt
    )
```

- 텍스트 데이터 **`임베딩`** → **`클러스터링`** → **각 클러스터에 대한 `요약`을 `생성`하는 과정** 수행함
  - **`클러스터링 진행`**
    - 주어진 `텍스트 목록`에 대해 `임베딩`을 `생성`하고 `유사성`에 `기반`한 클러스터링을 진행
    - `df_clusters` 데이터프레임을 결과 = `원본 텍스트`, `임베딩`, 그리고 `클러스터 할당 정보`가 포함됨
  - **`데이터프레임 항목 확장`**
    - 클러스터 할당을 쉽게 처리하기 위한 목적
    - 각 행은 `텍스트`, `임베딩`, `클러스터를 포함`하는 `새로운 데이터프레임`으로 `변환`
  - 확장된 데이터프레임에서 **`요약 생성`**
    - `고유한 클러스터 식별자`를 `추출`하고, 각 클러스터에 대한 `텍스트`를 `포맷팅`하여 요약 생성 → **`df_summary 데이터프레임`에 저장**
    - 각 클러스터의 `요약`, 지정된 `세부 수준`, 그리고 `클러스터 식별자`를 포함
  - 최종 반환
    - 함수 = **`두 개의 데이터프레임`을 포함하는 `튜플`을 반환**
    - 첫 번째 데이터프레임: **`원본 텍스트`**, **`임베딩`**, **`클러스터 할당 정보`**
    - 두 번째 데이터프레임: **`각 클러스터에 대한 요약`**, **`해당 세부 수준`**, **`클러스터 식별자`**

```
def embed_cluster_summarize_texts(
    texts: List[str], level: int
) -> Tuple[pd.DataFrame, pd.DataFrame]:
    """
    텍스트 목록에 대해 임베딩, 클러스터링 및 요약을 수행합니다. 이 함수는 먼저 텍스트에 대한 임베딩을 생성하고,
    유사성을 기반으로 클러스터링을 수행한 다음, 클러스터 할당을 확장하여 처리를 용이하게 하고 각 클러스터 내의 내용을 요약합니다.
```

```
    매개변수:
    - texts: 처리할 텍스트 문서 목록입니다.
    - level: 처리의 깊이나 세부 사항을 정의할 수 있는 정수 매개변수입니다.

    반환값:
    - 두 개의 데이터프레임을 포함하는 튜플:
        1. 첫 번째 데이터프레임(`df_clusters`)은 원본 텍스트, 그들의 임베딩, 그리고 클러스터 할당을 포함합니다.
        2. 두 번째 데이터프레임(`df_summary`)은 각 클러스터에 대한 요약, 지정된 세부 수준, 그리고 클러스터 식별자를 포함합니다.
    """

    # 텍스트를 임베딩하고 클러스터링하여 'text', 'embd', 'cluster' 열이 있는 데이터프레임 생성하기
    df_clusters = embed_cluster_texts(texts)

    # 클러스터를 쉽게 조작하기 위해 데이터프레임을 확장할 준비하기
    expanded_list = []

    # 데이터프레임 항목을 문서-클러스터 쌍으로 확장하여 처리를 간단하게 하기
    for index, row in df_clusters.iterrows():
        for cluster in row["cluster"]:
            expanded_list.append(
                {"text": row["text"], "embd": row["embd"], "cluster": cluster}
            )

    # 확장된 목록에서 새 데이터프레임 생성하기
    expanded_df = pd.DataFrame(expanded_list)

    # 처리를 위해 고유한 클러스터 식별자 검색하기
    all_clusters = expanded_df["cluster"].unique()

    print(f"--Generated {len(all_clusters)} clusters--")

    # 요약
    template = """여기 LangChain 표현 언어 문서의 하위 집합이 있습니다.

    LangChain 표현 언어는 LangChain에서 체인을 구성하는 방법을 제공합니다.

    제공된 문서의 자세한 요약을 제공하십시오.

    문서:
    {context}
    """
    prompt = ChatPromptTemplate.from_template(template)

    # llm 삽입
    chain = prompt | model | StrOutputParser()

    # 각 클러스터 내의 텍스트를 요약을 위해 포맷팅하기
    summaries = []
    for i in all_clusters:
        df_cluster = expanded_df[expanded_df["cluster"] == i]
        formatted_txt = fmt_txt(df_cluster)
        summaries.append(chain.invoke({"context": formatted_txt}))

    # 요약, 해당 클러스터 및 레벨을 저장할 데이터프레임 생성하기
    df_summary = pd.DataFrame(
        {
            "summaries": summaries,
            "level": [level] * len(summaries),
            "cluster": list(all_clusters),
        }
    )

    return df_clusters, df_summary                          # 1.1s
```

- 텍스트 데이터를 **재귀적으로** **임베딩**, **클러스터링** 및 **요약**하는 과정을 구현한 함수
  - 주어진 텍스트 리스트 = 임베딩, 클러스터링, 요약 → **각 단계별로** **결과 저장**
  - 함수: 최대 지정된 재귀 레벨까지 실행 or 유일한 클러스터의 수 = 1이 될 때까지 반복
  - 각 재귀 단계
    - 현재 레벨의 클러스터링 결과와 요약 결과를 데이터프레임 형태로 반환 → 이를 결과 딕셔너리에 저장
    - 만약 현재 레벨이 최대 재귀 레벨보다 작고, 유일한 클러스터의 수가 1보다 크다면: 현재 레벨의 요약 결과를 다음 레벨의 입력 텍스트로 사용하여 재귀적으로 함수를 호출
  - 최종 반환: **각 레벨별 클러스터 데이터프레임**, **요약 데이터프레임을 포함하는 딕셔너리**

```
def recursive_embed_cluster_summarize(
    texts: List[str], level: int = 1, n_levels: int = 3
) -> Dict[int, Tuple[pd.DataFrame, pd.DataFrame]]:

    """
    지정된 레벨까지 또는 고유 클러스터의 수가 1이 될 때까지 텍스트를 재귀적으로 임베딩, 클러스터링, 요약하여
```

```
    각 레벨에서의 결과를 저장합니다.

    매개변수:
    - texts: List[str], 처리할 텍스트들.
    - level: int, 현재 재귀 레벨 (1에서 시작).
    - n_levels: int, 재귀의 최대 깊이.

    반환값:
    - Dict[int, Tuple[pd.DataFrame, pd.DataFrame]], 재귀 레벨을 키로 하고 해당 레벨에서의 클러스터 DataFrame과 요약 DataFrame을 포함하는 튜플을 값
    """

    # 각 레벨에서의 결과를 저장할 사전
    results = {}

    # 현재 레벨에 대해 임베딩, 클러스터링, 요약 수행
    df_clusters, df_summary = embed_cluster_summarize_texts(texts, level)

    # 현재 레벨의 결과 저장
    results[level] = (df_clusters, df_summary)

    # 추가 재귀가 가능하고 의미가 있는지 결정
    unique_clusters = df_summary["cluster"].nunique()
    if level < n_levels and unique_clusters > 1:
        # 다음 레벨의 재귀 입력 텍스트로 요약 사용
        new_texts = df_summary["summaries"].tolist()

        next_level_results = recursive_embed_cluster_summarize(
            new_texts, level + 1, n_levels
        )

        # 다음 레벨의 결과를 현재 결과 사전에 병합
        results.update(next_level_results)

    return results                                              # 0.6s
```

```
# 전체 문서의 개수
len(docs_texts)
print(f"전체 문서의 개수: {len(docs_texts)} 개 ")
```

- 전체 문서의 개수: 231 개

```
# 트리 구축

import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"      # 병렬처리 비활성화

# 문서 텍스트를 리프 텍스트로 설정
leaf_texts = docs_texts

# 재귀적으로 임베딩, 클러스터링 및 요약을 수행하여 결과를 얻음
results = recursive_embed_cluster_summarize(
    leaf_texts,
    level=1,
    n_levels=1)                          # n_levels=3 → 1로 축소 (커널 충돌로 인한 지속된 오류)
```

```
print(results)
```

- `--Generated 24 clusters--` (`19m 37.3s`)

```
{1: (                                                text  \
0    \n\n\n\nLangChain Expression Language (LCEL) |...
1    \n\n\n\nCallbacks | ◆ü¶úÔ∏èüîó LangChain\n\n\...
2    \n\n\n\nHow to split by character | ◆ü¶úÔ∏èüî...
3    \n\n\n\nHow to load Microsoft Office files | ◆...
4    \n\n\n\nA Long-Term Memory Agent | ◆ü¶úÔ∏èüîó...
..                                                 ...
226  \n\n\n\nBuild a Question/Answering system over...
227  \n\n\n\nEmbedding models | ◆ü¶úÔ∏èüîó LangCha...
228  \n\n\n\nHow to get your RAG application to ret...
229  \n\n\n\n\nHow to use output parsers to parse a...
230  \n\n\n\n\nHow to do "self-querying" retrieval ...

                                             embd       cluster
0    [-0.08731497079133987, -0.03123190999031067, 0...  [16.0, 16.0]
1    [-0.09203468263149261, -0.04277104511857033, 0...        [20.0]
2    [-0.06461670994758606, -0.0202323105186224, 0....        [22.0]
3    [-0.0840206891298294, -0.02636653184890747, 0....         [5.0]
```

```
4    [-0.04305842146277428, -0.0423608161509037, -0...         [7.0]
..                                                 ...          ...
226  [-0.059163227677345276, -0.044757720082998276,...         [0.0]
227  [-0.07107416540384293, -0.07513609528541565, 0...        [10.0]
228  [-0.07376673072576523, -0.012838575057685375, ...         [0.0]
229  [-0.04497351124882698, -0.016761764883995056, ...  [23.0, 23.0]
230  [-0.04231127351522446, -0.018591172993183136, ...    [2.0, 2.0]

[231 rows x 3 columns],                                          summaries  level  cluster
0   LangChain 표현 언어(LCEL)는 LangChain에서 체인(chain)을 ...      1     16.0
1   제공된 문서는 LangChain의 **콜백(Callback) 시스템**에 대한 자세...      1     20.0
2   LangChain 표현 언어(LCEL)는 LangChain에서 체인(chains)을...      1     22.0
3   제공된 문서는 LangChain 표현 언어(LCEL) 자체에 대한 자세한 내용을 제...      1      5.0
4   제공된 문서 하위 집합에 따르면, **LangChain 표현 언어(LCEL)는 La...      1      7.0
5   제공된 문서들은 LangChain 표현 언어(LCEL)의 맥락에서 '예제 선택기(E...      1      1.0
6   제공된 문서에 따르면, **LangChain 표현 언어(LangChain Expre...      1     18.0
7   제공된 LangChain 표현 언어(LCEL) 문서는 LangChain에서 체인과 ...      1     12.0
8   LangChain 표현 언어(LCEL)는 LangChain 구성 요소를 사용하여 복...      1     19.0
9   제공된 문서는 LangChain 표현 언어(LangChain Expression L...      1      0.0
10  제공된 문서는 LangChain 표현 언어(LangChain Expression L...      1      4.0
11  LangChain 표현 언어(LCEL)는 LangChain에서 체인을 구성하는 방법...      1      2.0
12  LangChain 표현 언어(LCEL)는 LangChain에서 체인(chains)을...      1     10.0
13  LangChain 표현 언어(LCEL)는 LangChain에서 체인(chains)을...      1      9.0
14  LangChain 표현 언어(LCEL)는 LangChain에서 체인을 구성하는 방법...      1     21.0
15  제공된 문서는 LangChain 표현 언어(LCEL)의 하위 집합을 다루며, Lan...      1      8.0
16  LangChain 표현 언어(LCEL)는 LangChain에서 체인을 구성하는 핵심...      1     11.0
17  제공된 문서 하위 집합에 대한 자세한 요약은 다음과 같습니다.\n\nLangChai...      1     14.0
18  제공된 문서 하위 집합에 대한 자세한 요약은 다음과 같습니다.\n\nLangChai...      1     23.0
19  LangChain 표현 언어(LCEL)는 LangChain에서 체인을 구성하는 방법...      1     15.0
20  제공된 문서들은 LangChain 표현 언어(LCEL)의 하위 집합을 다루며, 주로...      1      6.0
21  제공된 문서는 'LangChain 표현 언어'에 대한 내용이라고 명시되어 있지만, ...      1     17.0
22  제공된 문서들은 'LangChain 표현 언어'(LCEL)를 탐색하는 섹션의 하위 ...      1      3.0
23  제공된 문서들은 LangChain 표현 언어(LCEL)의 역할과, LangChain...      1     13.0)}
```

```
print(type(results))        # <class 'dict'>
```

```
print(len(results))         # 1
```

- 논문: `collapsed tree retrieval` = 최고의 성능 보고
  - `트리 구조`를 `단일 계층`으로 `평탄화` → `모든 노드`에 대해 `동시에` `k-최근접 이웃`(`kNN`) 검색을 적용하는 과정 포함

- `Chroma` 벡터 저장소 사용 → 텍스트 데이터의 벡터화 및 검색 가능한 저장소 구축하는 과정
  - 초기: `leaf_texts`에 저장된 텍스트 데이터 = `all_texts` 변수에 복사
  - `결과 데이터`(`results`)를 `순회` → 각 레벨에서 `요약된 텍스트 추출` → `all_texts`에 `추가`
  - 각 레벨의 `DataFrame`에서 `summaries 컬럼의 값` = `리스트`로 `변환`하여 `추출` → `all_texts`에 `추가`
  - `모든 텍스트 데이터`(`all_texts`) 사용 → `Chroma` 벡터 저장소 구축
    - `Chroma.from_texts()` → `텍스트 데이터 벡터화`, `벡터 저장소 생성`
    - `.as_retriever()` → 검색기(retriever)를 초기화 → 생성된 벡터 저장소를 검색 가능하게 하기 위한 목적

- `result`

```
from langchain_community.vectorstores import FAISS

# leaf_texts 복사 → all_texts 초기화
all_texts = leaf_texts.copy()

# 각 레벨의 요약 추출 → all_texts에 추가하기 위해 결과를 순회하기
for level in sorted(results.keys()):
    # 현재 레벨의 DataFrame에서 요약 추출하기
    summaries = results[level][1]["summaries"].tolist()
    # 현재 레벨의 요약을 all_texts에 추가하기
    all_texts.extend(summaries)

# all_texts → FAISS vectorstore 구축하기
```

```
vectorstore = FAISS.from_texts(
    texts=all_texts,
    embedding=embd
    )                                          # 7.0s
```

- **DB**를 로컬에 저장하기

```
import os

DB_INDEX = "RAPTOR3"

# 로컬에 FAISS DB 인덱스가 이미 존재하는지 확인 → 그렇다면 로드하여 vectorstore와 병합한 후 저장하기
if os.path.exists(DB_INDEX):
    local_index = FAISS.load_local(DB_INDEX, embd)
    local_index.merge_from(vectorstore)
    local_index.save_local(DB_INDEX)

else:
    vectorstore.save_local(folder_path=DB_INDEX)
```

```
# retriever 생성
retriever3 = vectorstore.as_retriever()
```

- **RAG 체인 정의** → 특정 코드 예제를 요청하는 방법 구현하기
  - **hub.pull** → RAG 프롬프트 불러오기
  - **format_docs()**
    - 문서 포맷팅을 위해 정의한 함수
    - 이 함수는 문서의 페이지 내용을 연결하여 반환
  - **RAG Chain** 구성
    - 검색기 (retriever)로부터 문맥 가져오기 → format_docs 함수로 포맷팅 → 질문 처리
  - RunnablePassthrough() → 질문을 그대로 전달
  - **체인** : 프롬프트, 모델, StrOutputParser() → **최종 출력** = **문자열**로 **파싱**
  - **rag_chain.invoke** 메소드 사용 → **"How to define a RAG chain? Give me a specific code example."** 라는 질문 처리

```
# 모델(LLM) 생성하기
from langchain.callbacks.base import BaseCallbackHandler
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv
import os

load_dotenv()

GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY2")

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,                                          # temperature = 0으로 설정
    max_output_tokens=4096,
)

class StreamCallback(BaseCallbackHandler):
    def on_llm_new_token(self, token: str, **kwargs):
        print(token, end="", flush=True)
```

- LLM 생성하기

```
E0000 00:00:1759936339.865590 1301213 alts_credentials.cc:93] ALTS creds ignored. Not running on GCP and untruste
```

```
from langchain import hub
from langchain_core.runnables import RunnablePassthrough

# 프롬프트 생성
prompt = hub.pull("rlm/rag-prompt")

# 문서 포스트 프로세싱
```

```
def format_docs(docs):
    # 문서의 페이지 내용을 이어붙여 반환
    return "\n\n".join(doc.page_content for doc in docs)


# RAG 체인 정의
rag_chain3 = (
    # 검색 결과를 포맷팅하고 질문을 처리
    {"context": retriever3 | format_docs, "question": RunnablePassthrough()}
    | prompt                                              # 프롬프트 적용
    | gemini_lc                                           # 모델 적용
    | StrOutputParser()                                   # 문자열 출력 파서 적용
)                                                         # 0.3s
```

```
# Low Level 질문 실행_1
print(rag_chain3.invoke("RunnableParallel에 대한 예시 코드를 보여주세요."))
```

- `Low Level 질문 실행_1` (`6.4s`)

`RunnableParallel`은 여러 `Runnable`을 병렬로 실행하고 그 결과를 맵핑으로 반환합니다. 다음은 `RunnableLambda`를 사용하여 숫자에 1을 더

```
from langchain_core.runnables import RunnableLambda, RunnableParallel

def add_one(x: int) -> int: return x + 1
def mul_two(x: int) -> int: return x * 2

sequence = RunnableLambda(add_one) | RunnableParallel({"mul_two": RunnableLambda(mul_two)})
print(sequence.invoke(1)) # {'mul_two': 4}
```

```
# Low Level 질문 실행_2
print(rag_chain3.invoke("self-querying 방법과 예시 코드를 작성해 주세요."))
```

- `Low Level 질문 실행_2` (`12.9s`)

셀프 쿼리(Self-querying)는 LLM 체인을 사용하여 자연어 쿼리를 구조화된 쿼리로 변환하고, 이를 벡터 스토어에 적용하여 문서 내용의 의미론적 유사성

```
from langchain_chroma import Chroma
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings
from langchain.chains.query_constructor.schema import AttributeInfo
from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain_openai import ChatOpenAI

# 1. 문서 준비 및 벡터 스토어 생성
docs = [
    Document(page_content="A bunch of scientists bring back dinosaurs and mayhem breaks loose", metadata={"ye
    Document(page_content="Leo DiCaprio gets lost in a dream within a dream within a dream within a ...", met
    Document(page_content="A psychologist / detective gets lost in a series of dreams within dreams within dr
]
vectorstore = Chroma.from_documents(docs, OpenAIEmbeddings())

# 2. 메타데이터 정보 정의
metadata_field_info = [
    AttributeInfo(name="genre", description="The genre of the movie", type="string"),
    AttributeInfo(name="year", description="The year the movie was released", type="integer"),
    AttributeInfo(name="director", description="The name of the movie director", type="string"),
    AttributeInfo(name="rating", description="A 1-10 rating for the movie", type="float"),
]
document_content_description = "Brief summary of a movie"
llm = ChatOpenAI(temperature=0)

# 3. SelfQueryRetriever 인스턴스 생성
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectorstore,
    document_content_description,
    metadata_field_info,
```

```
)

# 4. 쿼리 실행 예시
print(retriever.invoke("I want to watch a movie rated higher than 8.5"))

# 출력 예시: [Document(page_content='A psychologist / detective gets lost in a series of dreams within dreams w
```

```
# Low Level 질문 실행_3
print(rag_chain3.invoke("Output parsers의 개념과 주요 메서드를 설명해주세요."))
```

- `Low Level 질문 실행_3` (`3.5s`)

```
Output 파서는 언어 모델의 텍스트 응답을 구조화된 형식으로 변환하는 데 도움을 주는 클래스입니다. 필수적으로 구현해야 하는 두 가지 주요 메서드는 언
```

```
# Low Level 질문 실행_3.2
print(rag_chain3.invoke("PydanticOutputParser로 시작하는 코드 예제를 안내해주세요."))
```

- `Low Level 질문 실행_3.2` (`5.3s`)

```
from langchain_core.output_parsers import PydanticOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import OpenAI
from pydantic import BaseModel, Field, model_validator

model = OpenAI(model_name="gpt-3.5-turbo-instruct", temperature=0.0)


# Define your desired data structure.
class Joke(BaseModel):
    setup: str = Field(description="question to set up a joke")
    punchline: str = Field(description="answer to resolve the joke")

    # You can add custom validation logic easily with Pydantic.
    @model_validator(mode="before")
    @classmethod
    def question_ends_with_question_mark(cls, values: dict) -> dict:
        setup = values.get("setup")
        if setup and setup[-1] != "?":
            raise ValueError("Badly formed question!")
        return values


# Set up a parser + inject instructions into the prompt template.
parser = PydanticOutputParser(pydantic_object=Joke)

prompt = PromptTemplate(
    template="Answer the user query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)

# And a query intended to prompt a language model to populate the data structure.
prompt_and_model = prompt | model
output = prompt_and_model.invoke({"query": "Tell me a joke."})
parser.invoke(output)
```

- next: ***05. 대화내용을 기억하는 RAG 체인***
