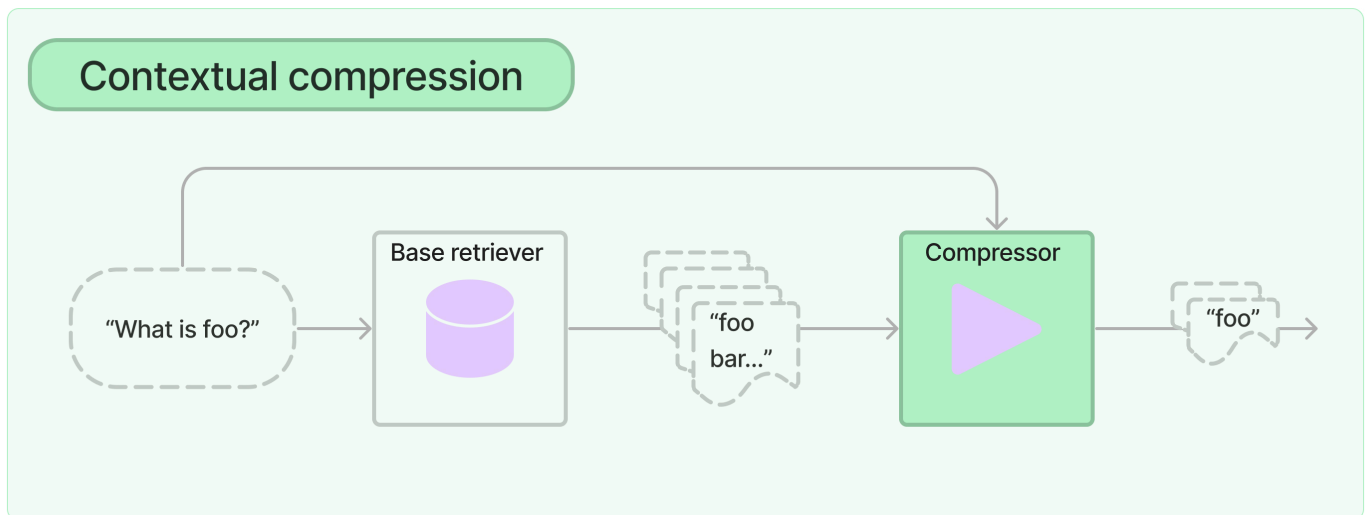


- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: <https://smith.langchain.com/hub/teddynote/summary-stuff-documents>

2. ContextualCompressionRetriever

1) 문맥 압축 검색기

- 검색 시스템의 어려움과 문제점
 - 사용자가 어떤 질문 (질의)을 할지 미리 알 수 없다는 점
 - 문제점: 질문에 가장 관련 있는 정보가 매우 많은 불필요한 텍스트가 담긴 문서 속에 섞여 있을 수 있음
 - 결과: 전체 문서를 언어 모델 (LLM)에 그대로 전달 시 → 비용이 많이 들고, 답변의 품질이 낮아짐



- 출처: <https://drive.google.com/uc?id=1CtNgWODXZudxAWSRiWgSGEoTNRUFT98v>

ContextualCompressionRetriever = 해결책

- 기본 아이디어: 검색된 문서를 바로 반환 X → 사용자의 질문 (질의)의 맥락을 사용 → 문서의 내용을 압축
- 압축의 의미:
 - 개별 문서에서 관련 없는 내용을 제거 → 내용을 줄임

- 불필요한 문서를 **목록**에서 **제외**하는 것을 모두 포함
- **작동 방식**:
 - 질의를 **기본 검색기 (base retriever)**에 전달
 - 기본 검색기에서 초기 문서를 가져옴
 - 이 문서를 **Document Compressor**에 통과시켜 가장 관련 있는 정보만 남도록 내용을 줄이거나 문서를 제거
- **목표**: 관련 있는 정보만 응용 프로그램에 전달되게 하여, LLM 호출 비용을 줄이고 답변 품질을 높이는 것

2) 설정

```
# 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"

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.

문서를 예쁘게 출력하기 위한 도우미 함수

```
def pretty_print_docs(docs):
    print(
        f"\n{'-' * 100}\n".join(
            [f"문서 {i+1}: \n\n" + d.page_content for i, d in enumerate(docs)]
        )
    )
```

3) 기본 Retriever 설정

- 간단한 벡터 스토어 `retriever` 초기화 → 텍스트 문서를 청크 단위로 저장하는 것부터 시작
- 예시 질문: `retriever` 는 관련 있는 문서 1~2 개와 관련 없는 문서 몇 개를 반환하는 것을 확인할 수 있음

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

1단계: Fake Embeddings 사용

```
embeddings = embeddings
```

임베딩 차원 크기를 계산

```
dimension_size = len(embeddings.embed_query("hello world"))
print(dimension_size)
print("✅ HuggingFaceEmbeddings 초기화 완료!")
```

384

✅ Hi

2단계: 문서 로더 및 분할

```
loader = TextLoader("../10_Retriever/data/appendix-keywords.txt")
documents = loader.load()
```

```
text_splitter = CharacterTextSplitter(
```

```

        chunk_size=300,
        chunk_overlap=0
    )
    split_docs = text_splitter.split_documents(documents) # split_document

```

3단계: 벡터스토어 생성

```
db = FAISS.from_documents(split_docs, embeddings)
```

4단계: 검색기(Retriever) 생성

```
retriever = db.as_retriever() # 벡터스토어에서 as_retriever() 호출
```

5단계: 검색 실행

```
docs = retriever.invoke("Semantic Search 에 대해서 알려줘.")
```

6단계: 결과 출력

```

def pretty_print_docs(docs):
    for i, doc in enumerate(docs):
        print(f"Document {i+1}:")
        print(doc.page_content)
        print("=" * 50)

pretty_print_docs(docs)
print("🎉✅ HuggingFaceEmbeddings로 완벽 실행!")

```

- 셀 출력

Document 1:

Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환함

예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환함

연관키워드: 자연어 처리, 검색 알고리즘, 데이터 마이닝

Embedding

=====

Document 2:

정의: 토큰라이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리에서 데이터를 전처리하는

예시: "I love programming."이라는 문장을 ["I", "love", "programming", "."]으로 분할함

연관키워드: 토큰화, 자연어 처리, 구문 분석

VectorStore

=====

Document 3:

정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽기

예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.

연관키워드: 데이터 교환, 웹 개발, API

Transformer

=====

Document 4:

정의: SQL(Structured Query Language)은 데이터베이스에서 데이터를 관리하기 위한 프로그래밍 언어입니다.

예시: `SELECT * FROM users WHERE age > 18;`은 18세 이상의 사용자 정보를 조회합니다.

연관키워드: 데이터베이스, 쿼리, 데이터 관리

CSV

=====

🎉 ✅ HuggingFaceEmbeddings로 완벽 실행!

▼ 4) 맥락적 압축 (Contextual Compression)

- LLM Chain Extractor → 생성한 DocumentCompressor를 retriever를 적용한 것 = ContextualCompressionRetriever

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain_google_genai import ChatGoogleGenerativeAI

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

# LLM을 사용하여 문서 압축기 생성
compressor = LLMChainExtractor.from_llm(gemini_lc)

# Contextual Compression Retriever 생성
compression_retriever = ContextualCompressionRetriever(
    # 문서 압축기와 리트리버를 사용하여 컨텍스트 압축 리트리버 생성
    base_compressor=compressor,
    base_retriever=retriever,
)

# 비교 실행
def pretty_print_docs(docs):
    for i, doc in enumerate(docs):
        print(f"Document {i+1}:")
        print(doc.page_content)
        print("=" * 50)
        print()

print("🔍 기본 Retriever 결과:")
basic_docs = retriever.invoke("Semantic Search 에 대해서 알려줘.")
```

```
pretty_print_docs(basic_docs)
```

```
print("=" * 60)
print("🧠 LLMChainExtractor 압축 후 결과:")
print("=" * 60)
```

```
compressed_docs = compression_retriever.invoke("Semantic Search 에 대해서 알려줘.")
pretty_print_docs(compressed_docs)
```

```
print("✅ Contextual Compression 완료!")
```

- 셀 출력 (3.4s)

```
E0000 00:00:1759054106.473395 2024367 alts_credentials.cc:93] ALTS creds ignored
```

🔍 기본 Retriever 결과:

Document 1:

정의: InstructGPT는 사용자의 지시에 따라 특정한 작업을 수행하기 위해 최적화된 GPT 모델입니다. 이 모델은 다양한 작업을 수행할 수 있습니다.
예시: 사용자가 "이메일 초안 작성"과 같은 특정 지시를 제공하면, InstructGPT는 관련 내용을 기반으로 이메일 초안을 작성합니다.
연관키워드: 인공지능, 자연어 이해, 명령 기반 처리

Keyword Search

=====

Document 2:

정의: 구조화된 데이터는 정해진 형식이나 스키마에 따라 조직된 데이터입니다. 이는 데이터베이스, 스프레드시트, XML 파일 등 다양한 형태로 존재할 수 있습니다.
예시: 관계형 데이터베이스에 저장된 고객 정보 테이블은 구조화된 데이터의 예입니다.
연관키워드: 데이터베이스, 데이터 분석, 데이터 모델링

Parser

=====

Document 3:

정의: 페이지 랭크는 웹 페이지의 중요도를 평가하는 알고리즘으로, 주로 검색 엔진 결과의 순위를 결정하는 데 사용됩니다.
예시: 구글 검색 엔진은 페이지 랭크 알고리즘을 사용하여 검색 결과의 순위를 정합니다.
연관키워드: 검색 엔진 최적화, 웹 분석, 링크 분석

데이터 마이닝

=====

Document 4:

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

Word2Vec

=====

=====

🧠 LLMChainExtractor 압축 후 결과:

=====

✅ Contextual Compression 완료!

✓ 5) LLM을 활용한 문서 필터링

• LLM Chain Filter

- 초기에 검색된 문서 중 어떤 문서를 필터링하고, 어떤 문서를 반환할지 결정하기 위해 **LLM 체인** 을 사용
- 보다 단순하지만 강력한 압축기
- 문서 내용을 **변경** or **압축** 하지 않고 문서를 **선택적으로 반환**

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainFilter
from langchain_google_genai import ChatGoogleGenerativeAI

# LLM 사용해 LLM Chain Filter 객체 생성하기
_filter = LLMChainFilter.from_llm(gemini_lc)

compression_retriever = ContextualCompressionRetriever(
    # LLMChainFilter와 retriever를 사용하여 ContextualCompressionRetriever 객체 생성하기
    base_compressor=_filter,
    base_retriever=retriever,
)

compressed_docs = compression_retriever.invoke(
    # 쿼리
    "Semantic Search 에 대해서 알려줘."
)

# 압축된 문서를 예쁘게 출력해보기
pretty_print_docs(compressed_docs)
```

• 셀 출력 (1.4s)

Document 1:

정의: 임베딩은 단어나 문장 같은 텍스트 데이터를 저차원의 연속적인 벡터로 변환하는 과정입니다. 이를 통해
예시: "사과"라는 단어를 [0.65, -0.23, 0.17]과 같은 벡터로 표현합니다.

연관키워드: 자연어 처리, 벡터화, 딥러닝

Token

=====

Document 2:

정의: FAISS는 페이스북에서 개발한 고속 유사성 검색 라이브러리로, 특히 대규모 벡터 집합에서 유사 벡터를
예시: 수백만 개의 이미지 벡터 중에서 비슷한 이미지를 빠르게 찾는 데 FAISS가 사용될 수 있습니다.

연관키워드: 벡터 검색, 머신러닝, 데이터베이스 최적화

Open Source

=====

- **EmbeddingFilter**

- **EmbeddingFilter**: 문서, 쿼리 임베딩 → 쿼리와 충분히 유사한 임베딩을 가진 문서만 반환
 - 더 저렴, 빠른 옵션 제공
 - 검색 결과의 관련성을 유지하면서도 계산 비용과 시간을 절약할 수 있음
- **EmbeddingsFilter** + **ContextualCompressionRetriever**
 - **EmbeddingsFilter** 사용, 유사도 임계값 (0.86) 이상인 문서 필터링

```
from langchain.retrievers.document_compressors import EmbeddingsFilter
from langchain_huggingface import HuggingFaceEmbeddings
```

```
# 1단계: 임베딩 초기화
embeddings = embeddings
```

```
print("✅ HuggingFaceEmbeddings 초기화 완료")
```

```
# ✅ Hugging
```

```
# 2단계: EmbeddingsFilter 객체 생성하기
embeddings_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=0.86
)
```

```
# 임베딩 모델
# 유사도 임계값
```

```
# 3단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=embeddings_filter,
    base_retriever=retriever
)
```

```
# 기본 압축기 =
# 기본 검색기 =
```

```
# 4단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기
compressed_docs = compression_retriever.invoke(
    # 쿼리
    "Semantic Search 에 대해서 알려줘."
)
```

```
# 5단계: 검색된 문서를 예쁘게 출력하기
```

```
pretty_print_docs(compressed_docs)
```


- 차원이 낮은 임베딩 모델, 높은 유사도 임계값 → 값이 출력되지 않음
- 유사도 임계값을 낮게 설정해서 다시 시도

```
from langchain.retrievers.document_compressors import EmbeddingsFilter
from langchain_core.embeddings import FakeEmbeddings

# 1단계: 허깅페이스 Embeddings 사용
embeddings = embeddings

# 2단계: EmbeddingsFilter 객체 생성하기
embeddings_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=0.01
)

# 3단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=embeddings_filter,
    base_retriever=retriever
)

# 4단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기
compressed_docs = compression_retriever.invoke(
    "Semantic Search 에 대해서 알려줘."
)

# 5단계: 검색된 문서를 예쁘게 출력하기
pretty_print_docs(compressed_docs)
```

- 셀 출력

Document 1:

정의: SQL(Structured Query Language)은 데이터베이스에서 데이터를 관리하기 위한 프로그래밍 언어임

예시: SELECT * FROM users WHERE age > 18;은 18세 이상의 사용자 정보를 조회합니다.

연관키워드: 데이터베이스, 쿼리, 데이터 관리

CSV

=====

```
from langchain.retrievers.document_compressors import EmbeddingsFilter
from langchain_core.embeddings import FakeEmbeddings

# 1단계: Fake Embeddings 사용
embeddings = FakeEmbeddings(size=384)

# 2단계: EmbeddingsFilter 객체 생성하기
embeddings_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=0.01
)
```

```
# 3단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=embeddings_filter,
    base_retriever=retriever
)

# 4단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기
compressed_docs = compression_retriever.invoke(
    "Semantic Search 에 대해서 알려줘."
)

# 5단계: 검색된 문서를 예쁘게 출력하기
pretty_print_docs(compressed_docs)
```

- 셀 출력 (최대 유사도 임계값 = 0.08)

Document 1:

정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형식으로, 사람과 기계 모두에게 읽기 쉽다.

예시: {"이름": "홍길동", "나이": 30, "직업": "개발자"}는 JSON 형식의 데이터입니다.

연관키워드: 데이터 교환, 웹 개발, API

Transformer

=====

Document 2:

정의: 토큰나이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리에서 데이터를 전처리하는 데 사용됩니다.

예시: "I love programming."이라는 문장을 ["I", "love", "programming", "."]으로 분할합니다.

연관키워드: 토큰화, 자연어 처리, 구문 분석

VectorStore

=====

6) 파이프라인 생성 (압축기 + 문서 변환기)

- **DocumentCompressionPipeline** → 여러 **compressor**를 순차적으로 결합 가능
 - **Compressor**와 함께 **BaseDocumentTransformer**를 파이프라인에 추가할 수 있음
 - 맥락적 압축을 수행하지 않고 단순한 문서 집합에 대한 변환 수행
 - **TextSplitter** = 더 작은 조각으로 분할하기 위해 **document transformer**로 사용 가능
 - **EmbeddingsRedundantFilter** = 문서 간 임베딩 유사성 (기본값 = 0.95 유사도 이상을 중복 문서로 간주)을 기반으로 중복 문서를 필터링하는 데 사용

- 순서: 문서를 더 작은 청크로 분할 → 중복 문서 제거 → 쿼리와 관련성 기준으로 필터링 →

`compressor pipeline` 생성

```
from langchain.retrievers.document_compressors import DocumentCompressorPipeline
from langchain_community.document_transformers import EmbeddingsRedundantFilter
from langchain_text_splitters import CharacterTextSplitter

# 1단계: 문자 기반 텍스트 분할기 생성하기
splitter = CharacterTextSplitter(
    chunk_size=300,                # 청크 크기 = 300
    chunk_overlap=0               # 청크 간 중복 X
)

# 2단계: 중복 필터 생성하기
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings) # 임베딩 사

# 3단계: 관련성 필터 생성하기
relevant_filter = EmbeddingsFilter(
    embeddings=embeddings,        # 임베딩 사용
    similarity_threshold=0.86     # 유사도 임계값을 0.86으로 설정
)

# 4단계: 문서 압축 파이프라인 생성하기
pipeline_compressor = DocumentCompressorPipeline(
    # 변환기로 설정하기
    transformers = [
        splitter,                # 텍스트 분할기
        redundant_filter,        # 중복 필터
        relevant_filter,         # 관련성 필터
        LLMChainExtractor.from_llm(gemini_lc) # LLM
    ]
)
```

- `ContextualCompressionRetriever` 초기화
- `base_compressor` = `pipeline_compressor`
- `base_retriever` = `retriever`

```
# 5단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor,
    base_retriever=retriever,
)
```

6단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기

```
compressed_docs = compression_retriever.invoke(
    "Semantic Search 에 대해서 알려줘." # 쿼리
)
```

7단계: 검색된 문서를 예쁘게 출력하기

```
pretty_print_docs(compressed_docs)
```

- 유사도 임계값의 차이로 인해 결과값이 나오지 않음
- 임베딩 모델들의 유사도 범위
 - OpenAI_embeddings의 유사도: 0.3 ~ 0.95
 - HuggingFace_embeddings의 유사도: 0.2 ~ 0.90
 - Google_Gemini의 유사도: 0.4 ~ 0.85
- 시도_1: 허깅페이스 임베딩 모델 사용

```
from langchain.retrievers.document_compressors import DocumentCompressorPipeline
from langchain_community.document_transformers import EmbeddingsRedundantFilter
from langchain_text_splitters import CharacterTextSplitter

# 1단계: 문자 기반 텍스트 분할기 생성하기
splitter = CharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=0
)

# 2단계: 중복 필터 생성하기
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)

# 3단계: 관련성 필터 생성하기
relevant_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=0.009
)

# 4단계: 문서 압축 파이프라인 생성하기
pipeline_compressor = DocumentCompressorPipeline(
    # 변환기로 설정하기
    transformers = [
        splitter,
        redundant_filter,
        relevant_filter,
        LLMChainExtractor.from_llm(gemini_lc)
    ]
)

# 5단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor,
    base_retriever=retriever,
)

# 6단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기
compressed_docs = compression_retriever.invoke(
    "Semantic Search 에 대해서 알려줘."
)
```

```
# 7단계: 검색된 문서를 예쁘게 출력하기
pretty_print_docs(compressed_docs)
```

- 셀 출력 (2.6s)

Document 1:

Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환함
예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환함
연관키워드: 자연어 처리, 검색 알고리즘, 데이터 마이닝

=====

- 시도_2: 허깅페이스 fake 임베딩 모델 사용

```
from langchain.retrievers.document_compressors import DocumentCompressorPipeline
from langchain_community.document_transformers import EmbeddingsRedundantFilter
from langchain_text_splitters import CharacterTextSplitter
from langchain_core.embeddings import FakeEmbeddings

# 1단계: 문자 기반 텍스트 분할기 생성하기
splitter = CharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=0
)

# 2단계: 중복 필터 생성하기
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)

# 3단계: 관련성 필터 생성하기
relevant_filter = EmbeddingsFilter(
    embeddings=FakeEmbeddings(size=384),
    similarity_threshold=0.009
)

# 4단계: 문서 압축 파이프라인 생성하기
pipeline_compressor = DocumentCompressorPipeline(
    # 변환기로 설정하기
    transformers = [
        splitter,
        redundant_filter,
        relevant_filter,
        LLMChainExtractor.from_llm(gemini_lc)
    ]
)

# 5단계: ContextualCompressionRetriever 객체 생성하기
compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor,
    base_retriever=retriever,
```

```
)
```

```
# 6단계: ContextualCompressionRetriever 객체를 사용하여 관련 문서 검색하기
compressed_docs = compression_retriever.invoke(
    "Semantic Search 에 대해서 알려줘."
)
```

```
# 쿼리
```

```
# 7단계: 검색된 문서를 예쁘게 출력하기
pretty_print_docs(compressed_docs)
```

- 셀 출력 (1.3s)

Document 1:

Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환하는 검색 방법이다.
예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환한다.
연관키워드: 자연어 처리, 검색 알고리즘, 데이터 마이닝

=====

7) 유동적 유사도 분포 분석

- 실시간 유사도 분포 분석해보기

```
import numpy as np
import matplotlib.pyplot as plt
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.retrievers.document_compressors import EmbeddingsFilter
from typing import List, Tuple
import warnings
warnings.filterwarnings("ignore")

class SimilarityThresholdCalculator:
    """유사도 임계값 자동 계산기"""

    def __init__(self, embeddings, retriever):
        self.embeddings = embeddings
        self.retriever = retriever

    def analyze_similarity_distribution(self, query: str, top_k: int = 10) -> dict:
        """쿼리에 대한 유사도 분포 분석"""

        print(f"🔍 쿼리 분석: '{query}'")

        # 기본 검색 결과
        docs = self.retriever.invoke(query)
        if not docs:
```

```

        print("❌ 검색된 문서가 없습니다.")
        return {}

# 쿼리 임베딩
query_embedding = self.embeddings.embed_query(query)

# 각 문서의 유사도 계산
similarities = []
for doc in docs:
    doc_embedding = self.embeddings.embed_documents([doc.page_content])[0]

    # 코사인 유사도 계산
    similarity = self._cosine_similarity(query_embedding, doc_embedding)
    similarities.append(similarity)

    print(f" 📄 유사도: {similarity:.4f} | {doc.page_content[:50]}...")

# 통계 계산
similarities = np.array(similarities)
stats = {
    'similarities': similarities,
    'mean': np.mean(similarities),
    'std': np.std(similarities),
    'min': np.min(similarities),
    'max': np.max(similarities),
    'median': np.median(similarities),
    'q25': np.percentile(similarities, 25),
    'q75': np.percentile(similarities, 75)
}

print(f"\n📊 유사도 통계:")
print(f"   평균: {stats['mean']:.4f}")
print(f"   표준편차: {stats['std']:.4f}")
print(f"   최소값: {stats['min']:.4f}")
print(f"   최대값: {stats['max']:.4f}")
print(f"   중간값: {stats['median']:.4f}")

return stats

def _cosine_similarity(self, vec1: List[float], vec2: List[float]) -> float:
    """코사인 유사도 계산"""
    vec1, vec2 = np.array(vec1), np.array(vec2)
    dot_product = np.dot(vec1, vec2)
    norms = np.linalg.norm(vec1) * np.linalg.norm(vec2)
    return dot_product / norms if norms != 0 else 0

def suggest_optimal_threshold(self, query: str, target_retention: float = 0.5)
    """최적 임계값 제안 (target_retention: 유지하고 싶은 문서 비율)"""

    stats = self.analyze_similarity_distribution(query)
    if not stats:
        return 0.01

    similarities = stats['similarities']

# 방법 1: 분위수 기반
threshold_percentile = (1 - target_retention) * 100

```

```

threshold_quantile = np.percentile(similarities, threshold_percentile)

# 방법 2: 평균 - n*표준편차 기반
threshold_std = stats['mean'] - 0.5 * stats['std']

# 방법 3: 안전 임계값 (최소값의 80%)
threshold_safe = stats['min'] * 0.8

# 세 방법 중 중간값 선택
candidates = [threshold_quantile, threshold_std, threshold_safe]
optimal_threshold = np.median(candidates)

# 최소 0.001, 최대 0.95 제한
optimal_threshold = max(0.001, min(0.95, optimal_threshold))

print(f"\n🎯 추천 임계값:")
print(f"  분위수 기반: {threshold_quantile:.4f}")
print(f"  표준편차 기반: {threshold_std:.4f}")
print(f"  안전 기반: {threshold_safe:.4f}")
print(f"  ✅ 최종 추천: {optimal_threshold:.4f}")

return optimal_threshold

def test_threshold_performance(self, query: str, thresholds: List[float]) ->
    """다양한 임계값 성능 테스트"""

    base_docs = self.retriever.invoke(query)
    base_count = len(base_docs)

    print(f"\n📝 임계값 성능 테스트 (기준: {base_count}개 문서)")
    print("="*60)

    results = {}

    for threshold in thresholds:
        try:
            # EmbeddingsFilter 생성
            embeddings_filter = EmbeddingsFilter(
                embeddings=self.embeddings,
                similarity_threshold=threshold
            )

            # 필터링 테스트
            filtered_docs = embeddings_filter.compress_documents(base_docs, c
            filtered_count = len(filtered_docs)
            retention_rate = filtered_count / base_count if base_count > 0 el

            results[threshold] = {
                'filtered_count': filtered_count,
                'retention_rate': retention_rate
            }

            print(f"임계값 {threshold:6.3f}: {filtered_count:2d}개 ({retention_

        except Exception as e:
            print(f"임계값 {threshold:6.3f}: ❌ 오류 - {e}")
            results[threshold] = {'filtered_count': 0, 'retention_rate': 0}

```



```

        return results

# 🎬 사용 예시
def auto_threshold_pipeline():
    """자동 임계값 계산 파이프라인"""

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

    # 기존 retriever 사용 (이미 생성된 것)
    # retriever = your_existing_retriever

    # 계산기 생성
    calc = SimilarityThresholdCalculator(embeddings, retriever)

    query = "Semantic Search 에 대해서 알려줘."

    # 1단계: 유사도 분포 분석
    stats = calc.analyze_similarity_distribution(query)

    # 2단계: 최적 임계값 제안
    optimal_threshold = calc.suggest_optimal_threshold(query, target_retention=0.5)

    # 3단계: 다양한 임계값 테스트
    test_thresholds = [0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, optimal_threshold]
    results = calc.test_threshold_performance(query, sorted(set(test_thresholds)))

    return optimal_threshold

# 실행
optimal_threshold = auto_threshold_pipeline()

```

- 셀 출력 (5.1s)

🔍 쿼리 분석: 'Semantic Search 에 대해서 알려줘.'

📄 유사도: 0.4923 | Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을...

📄 유사도: 0.3349 | 정의: 토큰라이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리

📄 유사도: 0.3063 | 정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형...

📄 유사도: 0.2875 | 정의: SQL(Structured Query Language)은 데이터베이스에서 데이터를 ...

📊 유사도 통계:

평균: 0.3553

표준편차: 0.0809

최소값: 0.2875


최대값: 0.4923


중간값: 0.3206


🔍 쿼리 분석: 'Semantic Search 에 대해서 알려줘.'

 유사도: 0.4923 | Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을...

 유사도: 0.3349 | 정의: 토큰라이저는 텍스트 데이터를 토큰으로 분할하는 도구입니다. 이는 자연어 처리

 유사도: 0.3063 | 정의: JSON(JavaScript Object Notation)은 경량의 데이터 교환 형...

 유사도: 0.2875 | 정의: SQL(Structured Query Language)은 데이터베이스에서 데이터를 ...

 유사도 통계:


평균: 0.3553

표준편차: 0.0809

최소값: 0.2875

최대값: 0.4923


중간값: 0.3206

 추천 임계값:

분위수 기반: 0.3120

표준편차 기반: 0.3148

안전 기반: 0.2300

 최종 추천: 0.3120

 임계값 성능 테스트 (기준: 4개 문서)

=====

임계값 0.001: 4개 (100.0%)

임계값 0.010: 4개 (100.0%)

임계값 0.050: 4개 (100.0%)

임계값 0.100: 4개 (100.0%)

임계값 0.200: 4개 (100.0%)

임계값 0.300: 3개 (75.0%)

임계값 0.312: 2개 (50.0%)

임계값 0.500: 0개 (0.0%)

임계값 0.700: 0개 (0.0%)

- 임계값 자동 설정 함수 만들어보기

```
from langchain.retrievers.document_compressors import DocumentCompressorPipeline,
from langchain_community.document_transformers import EmbeddingsRedundantFilter
from langchain_text_splitters import CharacterTextSplitter
from langchain.retrievers import ContextualCompressionRetriever

def create_auto_threshold_pipeline(embeddings, retriever, gemini_lc, query_sample):
    """자동 임계값 계산으로 파이프라인 생성"""

    print("🤖 자동 임계값 계산 시작...")

    # 1단계: 임계값 자동 계산
    calc = SimilarityThresholdCalculator(embeddings, retriever)
    optimal_threshold = calc.suggest_optimal_threshold(query_sample, target_reter

    print(f"✅ 계산된 최적 임계값: {optimal_threshold:.4f}")
```

```

# 2단계: 컴포넌트 생성
splitter = CharacterTextSplitter(chunk_size=300, chunk_overlap=0)
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)
relevant_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=optimal_threshold # 자동 계산된 값 사용!
)

# 3단계: 파이프라인 생성
try:
    from langchain.retrievers.document_compressors import LLMChainExtractor
    llm_extractor = LLMChainExtractor.from_llm(gemini_lc)

    pipeline_compressor = DocumentCompressorPipeline(
        transformers=[
            splitter,
            redundant_filter,
            relevant_filter,
            llm_extractor
        ]
    )
except ImportError:
    # LLMChainExtractor가 없으면 제외
    pipeline_compressor = DocumentCompressorPipeline(
        transformers=[
            splitter,
            redundant_filter,
            relevant_filter,
        ]
    )

# 4단계: 최종 검색기 생성
compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor,
    base_retriever=retriever,
)

print("🎉 자동 최적화 파이프라인 생성 완료!")
return compression_retriever, optimal_threshold

```

- 적응형 임계값 필터

```

class AdaptiveThresholdFilter:
    """임베딩 모델별 적응형 임계값 필터"""

    # 임베딩 모델별 기본 임계값 데이터베이스
    MODEL_THRESHOLDS = {
        'sentence-transformers/all-MiniLM-L6-v2': 0.15,
        'sentence-transformers/all-MiniLM-L12-v2': 0.20,
        'sentence-transformers/paraphrase-MiniLM-L3-v2': 0.12,
        'text-embedding-3-small': 0.75, # OpenAI
        'text-embedding-3-large': 0.80, # OpenAI
        'models/embedding-001': 0.70, # Google
        'fake': 0.01, # FakeEmbeddings
    }

```

```

}

def __init__(self, embeddings):
    self.embeddings = embeddings
    self.model_name = self._detect_model_name()

def _detect_model_name(self) -> str:
    """임베딩 모델명 감지"""
    embedding_type = str(type(self.embeddings))

    if 'Fake' in embedding_type:
        return 'fake'
    elif hasattr(self.embeddings, 'model_name'):
        return self.embeddings.model_name
    elif 'OpenAI' in embedding_type:
        return 'text-embedding-3-small'
    elif 'Google' in embedding_type:
        return 'models/embedding-001'
    else:
        return 'sentence-transformers/all-MiniLM-L6-v2' # 기본값

def get_recommended_threshold(self) -> float:
    """모델별 추천 임계값 반환"""
    threshold = self.MODEL_THRESHOLDS.get(self.model_name, 0.15)
    print(f"🌀 모델 '{self.model_name}' 추천 임계값: {threshold}")
    return threshold

def create_filter(self, custom_threshold: float = None) -> EmbeddingsFilter:
    """최적화된 EmbeddingsFilter 생성"""
    threshold = custom_threshold if custom_threshold else self.get_recommended_threshold()

    return EmbeddingsFilter(
        embeddings=self.embeddings,
        similarity_threshold=threshold
    )

# 📦 Jay를 위한 완벽한 사용법
def jay_optimized_pipeline(embeddings, retriever, gemini_lc):
    """Jay 전용 최적화 파이프라인"""

    print(f"🌀 Jay 전용 최적화 파이프라인 시작...")

    # 1단계: 적응형 임계값 계산
    adaptive_filter = AdaptiveThresholdFilter(embeddings)
    optimal_threshold = adaptive_filter.get_recommended_threshold()

    # 2단계: 실시간 검증 (선택사항)
    query_test = "Semantic Search 에 대해서 알려줘."
    calc = SimilarityThresholdCalculator(embeddings, retriever)

    # 빠른 검증
    test_results = calc.test_threshold_performance(query_test, [optimal_threshold])
    retention_rate = test_results[optimal_threshold]['retention_rate']

    # 임계값 조정 (보정)
    if retention_rate < 0.1:
        optimal_threshold *= 0.5 # 너무 적게 통과

```

```

        print(f"⚠ 임계값 너무 높음. 조정: {optimal_threshold:.4f}")
    elif retention_rate > 0.9:                                     # 너무 많이 통과
        optimal_threshold *= 1.5
        print(f"⚠ 임계값 너무 낮음. 조정: {optimal_threshold:.4f}")

    print(f"✅ 최종 최적화된 임계값: {optimal_threshold:.4f}")

# 3단계: 파이프라인 구성
splitter = CharacterTextSplitter(chunk_size=300, chunk_overlap=0)
redundant_filter = EmbeddingsRedundantFilter(embeddings=embeddings)
relevant_filter = EmbeddingsFilter(
    embeddings=embeddings,
    similarity_threshold=optimal_threshold
)

# LLM Extractor 추가 (가능하면)
transformers = [splitter, redundant_filter, relevant_filter]

try:
    from langchain.retrievers.document_compressors import LLMChainExtractor
    llm_extractor = LLMChainExtractor.from_llm(gemini_lc)
    transformers.append(llm_extractor)
    print("✅ LLM 압축기 추가됨")
except:
    print("⚠ LLM 압축기 제외 (기본 필터만 사용)")

# 4단계: 최종 파이프라인
pipeline_compressor = DocumentCompressorPipeline(transformers=transformers)

compression_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline_compressor,
    base_retriever=retriever,
)

return compression_retriever, optimal_threshold

# 📺 실행 예시
compression_retriever, threshold = jay_optimized_pipeline(embeddings, retriever,
print(f"🎉 최적화 완료! 사용된 임계값: {threshold}")

```

• 셀 출력

🎯 Jay 전용 최적화 파이프라인 시작...

🎯 모델 'fake' 추천 임계값: 0.01

📏 임계값 성능 테스트 (기준: 4개 문서)

=====

임계값 0.010: 2개 (50.0%)

✅ 최종 최적화된 임계값: 0.0100

✅ LLM 압축기 추가됨

🎉 최적화 완료! 사용된 임계값: 0.01

- 유사도 임계값 계산 자동화

```
def one_click_contextual_compression(embeddings, retriever, gemini_lc, query="Ser
    """원클릭 완전 자동화 시스템"""

    print("🚀 원클릭 자동 최적화 시작!")
    print("="*50)

    # 1단계: 임베딩 모델 분석
    model_type = str(type(embeddings))
    print(f"🔍 감지된 모델: {model_type}")

    # 2단계: 기본 임계값 추정
    if 'Fake' in model_type:
        base_threshold = 0.01
        print("✍️ FakeEmbeddings 감지 → 초저 임계값 모드")
    elif 'HuggingFace' in model_type:
        base_threshold = 0.15
        print("😊 HuggingFace 모델 감지 → 중간 임계값 모드")
    elif 'OpenAI' in model_type:
        base_threshold = 0.75
        print("🤖 OpenAI 모델 감지 → 고 임계값 모드")
    elif 'Google' in model_type:
        base_threshold = 0.70
        print("🌟 Google 모델 감지 → 고 임계값 모드")
    else:
        base_threshold = 0.20
        print("❓ 알 수 없는 모델 → 기본 임계값 모드")

    # 3단계: 실시간 검증 및 조정
    print(f"\n📝 임계값 {base_threshold} 검증 중...")

    # 테스트 쿼리로 성능 확인
    base_docs = retriever.invoke(query)
    if not base_docs:
        print("❌ 검색 결과 없음 - 기본 설정 사용")
        final_threshold = base_threshold
    else:
        # 임계값 테스트
        test_filter = EmbeddingsFilter(embeddings=embeddings, similarity_threshold=base_threshold)
        filtered_docs = test_filter.compress_documents(base_docs, query)
        retention_rate = len(filtered_docs) / len(base_docs)

        print(f"🇮🇹 테스트 결과: {len(base_docs)}개 → {len(filtered_docs)}개 ({reter

    # 동적 조정
    if retention_rate < 0.1:
        final_threshold = base_threshold * 0.3
        print(f"⬇️ 임계값 낮춤: {final_threshold:.4f}")
    elif retention_rate > 0.9:
        final_threshold = base_threshold * 1.5
        print(f"⬆️ 임계값 높임: {final_threshold:.4f}")
    else:
        final_threshold = base_threshold
        print(f"✅ 임계값 적정: {final_threshold:.4f}")
```

```

# 4단계: 파이프라인 자동 구성
print(f"\n🔧 파이프라인 구성 중...")

transformers = [
    CharacterTextSplitter(chunk_size=300, chunk_overlap=0),
    EmbeddingsRedundantFilter(embeddings=embeddings),
    EmbeddingsFilter(embeddings=embeddings, similarity_threshold=final_thresh
]

# LLM 압축기 추가 시도
try:
    from langchain.retrievers.document_compressors import LLMChainExtractor
    transformers.append(LLMChainExtractor.from_llm(gemini_lc))
    print("✅ LLM 압축기 추가")
except:
    print("⚠️ LLM 압축기 생략")

# 5단계: 최종 검색기 생성
pipeline = DocumentCompressorPipeline(transformers=transformers)
final_retriever = ContextualCompressionRetriever(
    base_compressor=pipeline,
    base_retriever=retriever
)

print(f"\n🎉 자동 최적화 완료!")
print(f"📊 최종 설정:")
print(f"  - 임계값: {final_threshold:.4f}")
print(f"  - 파이프라인 단계: {len(transformers)}개")
print("="*50)

return final_retriever, final_threshold

```

🚀 최종 사용법

- `def pretty_print_docs(docs):`
if docs:

🚀 원클릭 자동 최적화 시작!

=====

🔍 감지된 모델: <class 'langchain_core.embeddings.fake.FakeEmbeddings'>

📝 FakeEmbeddings 감지 → 초저 임계값 모드

🖋️ 임계값 0.01 검증 중...

📊 테스트 결과: 4개 → 2개 (50.0%)

✅ 임계값 적정: 0.0100

🔧 파이프라인 구성 중...

✅ LLM 압축기 추가

🎉 자동 최적화 완료!

📊 최종 설정:

– 임계값: 0.0100

– 파이프라인 단계: 4개

=====

🔍 최적화된 파이프라인 테스트:

Document 1:

Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환하는 방식이다. 예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환한다.

연관키워드: 자연어 처리, 검색 알고리즘, 데이터 마이닝

=====

- next: **앙상블 검색기 (EnsembleRetriever)**