RETRIEVER + (Hybrid Search)

날짜	@2026/01/26
구분	LangGraph를 활용한 AI Agent

▼ 1. Hybrid Search는 무엇인가?

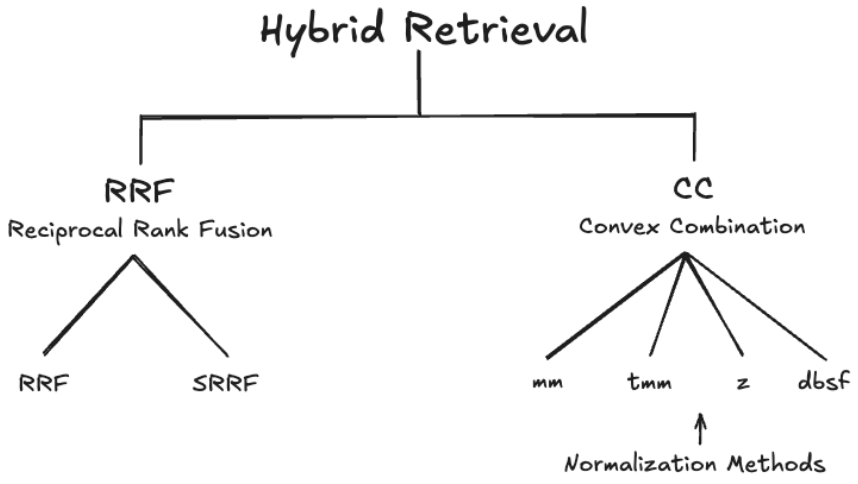
Dense Search와 **Sparse Search**를 함께 사용하여, Dense Search의 의미 기반 검색 장점과 Sparse Search의 정확한 키워드 매칭 장점을 동시에 활용하고, 각 방식의 단점을 보완하는 하이브리드 검색 방식

- **Dense Search**
  - 임베딩 모델을 사용하여, 문장을 고차원 벡터로 변환 유사도로 검색
  - 문맥과 의미를 반영한 검색으로, 유사한 내용을 잘 찾음
  - 특정 도메인 용어처럼 정확한 키워드 매칭이 필요한 경우 성능이 떨어짐
- **Sparse Search**
  - 텍스트를 단어 단위로 쪼개고, TF-IDF 기반으로 중요도를 계산하는 방식
  - 키워드 매칭에 강함
  - 동의어 및 오타 처리에 약하고, 의미적 유사도 반영이 어려움

즉, Hybrid Search는 도메인 용어는 Sparse Search로, 동의어나 일부 오타는 Dense Search로 처리하여 높은 정확도를 제공

▼ 2. Hybrid Search 결합 방식

Dense Search와 **Sparse Search**의 결과를 결합하는 방법은 크게 총 2가지가 존재



- **Convex Combination (CC)**
  - 각 문서에 대해 Dense Score와 Sparse Score를 정규화한 후, 가중합을 통해 최종 점수를 계산하는 방식

$$\text{Score}(d) = \alpha \cdot \tilde{S}_{\text{dense}}(d) + (1 - \alpha) \cdot \tilde{S}_{\text{sparse}}(d)$$

```
from langchain.retrievers import EnsembleRetriever
from langchain_community.vectorstores import OpenSearchVectorSearch

vector_db = OpenSearchVectorSearch(
            index_name=index_name,
            opensearch_url=oss_endpoint,
```

```
            embedding_function=embeddings,
            http_auth=http_auth,
            is_aoss =False,
            engine="faiss",
            space_type="l2"
        )

# Dense Retriever(유사도 검색)
opensearch_semantic_retriever = vector_db.as_retriever(
    search_type="similarity",
    search_kwargs={
        "k": 3
    }
)

search_semantic_result = opensearch_semantic_retriever.get_relevant_documents(quer
y)

opensearch_lexical_retriever = OpenSearchLexicalSearchRetriever(
    os_client=os_client,
    index_name=index_name
)

# Sparse Retriever
opensearch_lexical_retriever.update_search_params(
    k=3,
    minimum_should_match=0
)

search_keyword_result = opensearch_lexical_retriever.get_relevant_documents(query)

# Ensemble Retriever(앙상블 검색)
ensemble_retriever = EnsembleRetriever(
    retrievers=[opensearch_lexical_retriever, opensearch_semantic_retriever],
    weights=[0.50, 0.50]
)

search_hybrid_result = ensemble_retriever.get_relevant_documents(query)
```

- **Reciprocal Rank Fusion (RFF)**
  - 쿼리 실행 시 각 문서에 대해 순위 기반 결과를 반환하며, Dense와 Sparse 검색 결과를 각각 순위(rank)로 변환한 뒤, 순위의 역수를 합산하여 최종 점수를 계산하는 방식
  - 랭크 기반으로, 세밀한 가중치 조정이 어렵지만 반대로 별도의 스케일링 처리 안함

$$\mathrm{RRF}(d) = \sum_{s \in \{\mathrm{dense}, \mathrm{sparse}\}} \frac{1}{k + \mathrm{rank}_s(d)}$$

```
def rrf_fuse(rank_lists, k=60):
    score = {}
    for ranks in rank_lists:                        # 각 검색 결과(rank list)에 대해 반
복
        for r, doc_id in enumerate(ranks):          # r: 순위 (0부터 시작)
            score[doc_id] = score.get(doc_id, 0) + 1.0 / (k + r + 1)
    return sorted(score.items(), key=lambda x: x[1], reverse=True)
```

```
# 1) Dense / Sparse 각각 검색 수행
lex_docs = opensearch_lexical_retriever.get_relevant_documents(query)   # Sparse (B
M25 등)
sem_docs = opensearch_semantic_retriever.get_relevant_documents(query)  # Dense (Ve
ctor)

# 2) 문서 ID 기준 순위 리스트 생성
lex_ids = [d.metadata.get("id", d.page_content[:50]) for d in lex_docs]
sem_ids = [d.metadata.get("id", d.page_content[:50]) for d in sem_docs]

# 3) RRF 결합
fused = rrf_fuse([lex_ids, sem_ids], k=60)

# 4) 최종 문서 재구성
id2doc = {d.metadata.get("id", d.page_content[:50]): d for d in (lex_docs + sem_doc
s)}
final_docs = [id2doc[doc_id] for doc_id, _ in fused]
```

▼ 3. 구현 시 고려사항

- LangCahin은 OpeanSearch 기반 Sparse Retriever를 공식적으로 제공하지 않아, Custom Retriever 클래스를 직접 구현하고 오버라이딩해야함
- Dense Search와 Sparse Search를 각각 독립적으로 구성해야 하므로, 검색 파이프라인 복잡도 증가

앞서 확인한 부분에 대해서는 '어떤 문서를 가져올 것인가' 에 대해 다루었다면, 다음은 '얼마나 정확한 문서를 찾았는가' 에 대해 다뤄보려고 한다.

▼ 4. 어떻게 검색 품질을 향상시킬 수 있는가?

Hybrid Search를 통해 상위 *k*개의 후보 문서를 확보하더라도, 해당 결과가 반드시 최종 질의에 가장 근접한 문서 집합이라고 보기는 어려우며, 이유는 다음과 같음

- 상위 후보 문서 중 일부만 질문과 직접적인 관련이 있을 수 있음
- 법 조항, 특정 수치와 같은 핵심 정보가 상대적으로 랭킹 하위로 밀릴 수 있음
- Dense Search는 질의 의도와 다른 의미적 유사 문서를, Sparse Search는 문맥상 관련성이 낮은 키워드 일치 문서를 포함할 수 있어, Hybrid Search 결과에는 의미적·어휘적 노이즈가 혼재될 수 있음

▼ 5. ReRanker

품질 향상을 위해 쿼리에 대한 관련 도큐먼트가 컨텍스트에 존재할 뿐만 아니라, 컨텍스트 내에서의 순서 또한 상위권에 위치해야 하며, 이에 대한 해결책으로 ReRanker가 요구

- 기존 벡터 서치와 마찬가지로, Reranker 또한 질문(Query)과 문서(Document) 사이의 유사도 측정을 목표로 함
- 하지만, 기존 벡터 서치는 질문과 문서를 독립적으로 임베딩하는 Bi-encdoer 구조 사용
- Reranker는 질문과 문서를 하나의 인풋으로 처리([CLS] Query [SEP] Document [SEP])하는 Cross-encoder 구조로, 질문과 문서를 동시에 분석 (= Self-attention)