Day 5: Embedding & Vector DB - 완전 가이드

> **GitHub Repository**: RAG-on-Local-CPU-minimal

> 실제 문서 검색 - RAG의 핵심

🎯 학습 목표

Day 5의 핵심은 다음과 같습니다:

> **"임베딩으로 문서를 검색하고, 검색 결과를 LLM에 전달한다"**

이 Day를 마치면:

- ☑ SentenceTransformer로 임베딩 생성
- ☑ FAISS로 벡터 검색
- ☑ Ingest (문서 색인) 과정
- ☑ Top-K 검색 이해
- ☑ 왜 임베딩이 RAG의 핵심인지 이해

핵심: RAG = Retrieval (검색) + Generation (생성)

📂 프로젝트 구조

```
Rag_minimal_day5/
├─ main.py            # FastAPI 진입점
├─ pipeline.py        # RAG 파이프라인
├─ embedding.py       # 임베딩 생성
├─ vectorstore.py     # FAISS 벡터 검색
├─ ingest.py          # 문서 색인 (1회)
├─ loader.py          # 문서 로딩
├─ llm.py             # LLM Stub (더미)
├─ config.py          # 설정
└─ data/
   └─ docs.txt        # 입력 문서
```

🤔 왜 이런 구조인가?

1. 왜 임베딩이 필요한가?

키워드 검색의 한계:

```
질문: "강아지 키우는 방법"
문서1: "반려견 사육 가이드"  ← 검색 안됨!
문서2: "강아지 키우기 팁"    ← 검색됨
```

임베딩 검색:

```
질문: "강아지 키우는 방법"
  → 벡터: [0.2, 0.5, -0.3, ...]

문서1: "반려견 사육 가이드"
  → 벡터: [0.3, 0.4, -0.2, ...]  ← 유사! 검색됨

문서2: "강아지 키우기 팁"
  → 벡터: [0.25, 0.48, -0.28, ...] ← 유사! 검색됨
```

왜 더 좋은가?

- ☑ 의미 기반 검색
- ☑ 동의어 인식
- ☑ 언어 장벽 극복
- ☑ 문맥 이해

2. 왜 FAISS인가?

다른 Vector DB:

- Pinecone: 클라우드, 유료
- Weaviate: 설치 복잡
- Milvus: 무겁고 복잡
- Qdrant: 좋지만 설정 필요

FAISS 선택 이유:

- ☑ **CPU 전용**: GPU 불필요
- ☑ **메모리 기반**: 빠른 검색
- ☑ **Facebook 개발**: 검증됨
- ☑ **Python 바인딩**: 쉬운 사용
- ☑ **교육용 최적**: 단순 명확

실무 전환:

```
개발: FAISS (로컬)
  ↓
프로덕션: Pinecone/Weaviate (클라우드)
```

3. 왜 Ingest 단계가 필요한가?

실시간 임베딩 문제:

```
매 요청마다:
  문서 1000개 → 임베딩 → 검색
  시간: 5초
  ❌ 너무 느림!
```

Ingest 방식:

```
서버 시작 시 1회:
  문서 1000개 → 임베딩 → 저장
  시간: 5초 (한 번만)

매 요청:
  쿼리 → 임베딩 → 검색
  시간: 0.1초
  ☑ 빠름!
```

4. 왜 검색과 생성을 분리하나?

분리하지 않으면:

```python
def pipeline(query):
    # 검색 + 생성 혼재
    docs = search_and_generate(query)
    # 뭐가 검색이고 뭐가 생성인지 불명확
```

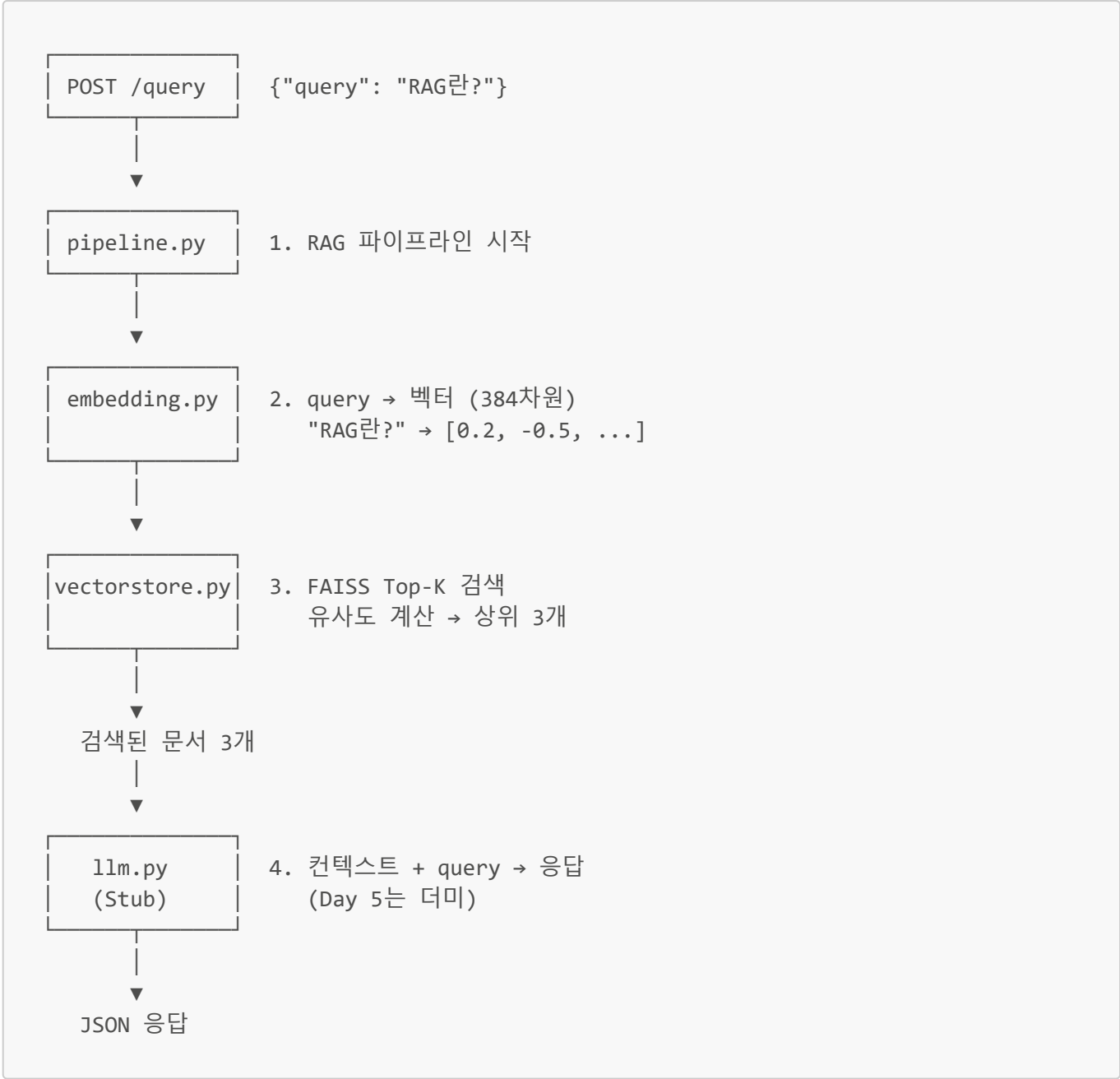
분리하면:

```python
def pipeline(query):
    # 1. 검색 (Retrieval)
    docs = vector_store.search(query, k=3)

    # 2. 생성 (Generation)
    response = llm.generate(docs, query)

    # 각 단계 독립 최적화 가능!
```

장점:

- ☑ 검색 알고리즘만 교체 가능
- ☑ LLM만 교체 가능
- ☑ 각각 테스트 가능
- ☑ 성능 병목 파악 쉬움

🔄 실행 흐름도



⚙ 설치 및 실행

1. 패키지 설치

Windows (PowerShell):

```
cd Rag_minimal_day5
pip install fastapi uvicorn sentence-transformers faiss-cpu
```

macOS/Linux (Bash):

```
cd Rag_minimal_day5
pip install fastapi uvicorn sentence-transformers faiss-cpu
```

설치 시간:

- sentence-transformers: 약 500MB
- 첫 실행 시 모델 다운로드: 약 90MB

2. 서버 실행

```
uvicorn main:app --reload
```

초기 로딩:

```
INFO: Loading embedding model...
INFO: all-MiniLM-L6-v2 loaded
INFO: Ingesting documents...
INFO: 10 documents indexed
INFO: Server ready
```

3. API 테스트

Swagger UI:

```
http://127.0.0.1:8000/docs
```

curl (PowerShell):

```
curl -X POST "http://127.0.0.1:8000/query" `
  -H "Content-Type: application/json" `
  -d '{"query":"RAG란 무엇인가?"}'
```

curl (Bash):

```
curl -X POST "http://127.0.0.1:8000/query" \
  -H "Content-Type: application/json" \
  -d '{"query":"RAG란 무엇인가?"}'
```

🧠 핵심 개념 Deep Dive

1. Embedding (임베딩)

텍스트 → 벡터 변환:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
text = "RAG는 검색 기반 생성이다"
vector = model.encode(text)

print(vector.shape)  # (384,)
print(vector[:5])    # [0.23, -0.45, 0.78, -0.12, 0.56]
```

384차원이란?

- 각 차원이 의미의 한 측면
- 차원 0: 기술 관련도
- 차원 1: 긍정/부정
- 차원 2: 추상/구체
- ...

유사도 계산:

```
vec1 = model.encode("강아지")
vec2 = model.encode("반려견")
vec3 = model.encode("자동차")

# 코사인 유사도
similarity(vec1, vec2)  # 0.85 (높음)
similarity(vec1, vec3)  # 0.12 (낮음)
```

2. FAISS Index

IndexFlatL2란?

```
import faiss

# 384차원 벡터용 인덱스
index = faiss.IndexFlatL2(384)

# 문서 벡터 추가
index.add(embeddings)  # (N, 384)

# 검색
query_vec = model.encode([query])  # (1, 384)
distances, indices = index.search(query_vec, k=3)
```

L2 거리:

```
L2(v1, v2) = sqrt(sum((v1[i] - v2[i])^2))

작을수록 유사함
거리 0.5: 매우 유사
거리 2.0: 보통
거리 5.0: 다름
```

3. Top-K 검색

K=3 예시:

```
질문: "RAG 개념"

문서1: "RAG는..." → 거리: 0.3  ← 선택
문서2: "Retrieval..." → 거리: 0.5  ← 선택
문서3: "LLM이란..." → 거리: 2.1
문서4: "Vector DB..." → 거리: 0.7  ← 선택
문서5: "날씨..." → 거리: 5.0
```

K 값 설정:

- K=1: 너무 적음, 정보 부족
- K=3: 적당 (기본)
- K=5: 조금 많음
- K=10: 너무 많음, 노이즈

4. Ingest 과정

단계별 과정:

```
# 1. 문서 로딩
docs = load_documents()  # ["doc1", "doc2", ...]

# 2. 임베딩 생성
embeddings = model.encode(docs)  # (N, 384)

# 3. 인덱스 저장
index.add(embeddings)

# 4. 메모리에 유지
# 서버가 죽으면 재생성 필요
```

메모리 vs 디스크:

```
# 메모리 (빠름, 휘발성)
index = faiss.IndexFlatL2(384)
```

```
# 디스크 저장 (느림, 영구)
faiss.write_index(index, "index.faiss")
index = faiss.read_index("index.faiss")
```

📖 코드 Deep Dive

config.py

```
# Line 1-3: 설정 상수
MODEL_NAME = "all-MiniLM-L6-v2"  # 임베딩 모델
TOP_K = 3                         # 검색할 문서 수
EMBEDDING_DIM = 384               # 벡터 차원
```

embedding.py

```
# Line 1-2: 임포트
from sentence_transformers import SentenceTransformer
from config import MODEL_NAME

# Line 4-5: 전역 모델 (1회 로딩)
_model = None

# Line 7-11: 모델 로딩 (싱글톤)
def get_model():
    global _model
    if _model is None:
        _model = SentenceTransformer(MODEL_NAME)
    return _model

# Line 13-15: 임베딩 생성
def embed(texts):
    model = get_model()
    return model.encode(texts)
```

핵심:

- 모델 1회만 로딩 (싱글톤 패턴)
- 매번 로딩하면 느림
- 메모리에 캐시

vectorstore.py

```
# Line 1-3: 임포트
import faiss
```

```
import numpy as np
from config import EMBEDDING_DIM, TOP_K

# Line 5-9: VectorStore 클래스
class VectorStore:
    def __init__(self):
        # L2 거리 기반 인덱스
        self.index = faiss.IndexFlatL2(EMBEDDING_DIM)
        self.documents = []  # 원본 문서 저장

    # Line 11-14: 문서 추가
    def add(self, embeddings, documents):
        # NumPy 배열로 변환
        embeddings = np.array(embeddings).astype('float32')
        self.index.add(embeddings)
        self.documents.extend(documents)

    # Line 16-21: 검색
    def search(self, query_embedding, k=TOP_K):
        # (1, 384) → (1, 384) 확인
        query_vec = np.array([query_embedding]).astype('float32')

        # FAISS 검색
        distances, indices = self.index.search(query_vec, k)

        # 결과 반환
        return [
            {
                "document": self.documents[i],
                "distance": float(distances[0][j])
            }
            for j, i in enumerate(indices[0])
        ]
```

핵심:

- `IndexFlatL2`: 정확한 검색 (느리지만 완벽)
- `astype('float32')`: FAISS 요구사항
- 원본 문서도 함께 저장

ingest.py

```
# Line 1-4: 임포트
from loader import load_documents
from embedding import embed
from vectorstore import VectorStore

# Line 6-15: Ingest 함수
def ingest():
    print("📂 Loading documents...")
    docs = load_documents()
```

```
    print("🔢 Generating embeddings...")
    embeddings = embed(docs)

    print("💾 Building vector store...")
    store = VectorStore()
    store.add(embeddings, docs)

    print(f"☑️ Indexed {len(docs)} documents")
    return store
```

pipeline.py

```
# Line 1-5: 임포트
from embedding import embed
from llm import generate

# Line 7-10: 전역 변수
_vector_store = None

# Line 12-15: 초기화
def init(store):
    global _vector_store
    _vector_store = store

# Line 17-30: RAG 파이프라인
def pipeline(query):
    # 1. Query 임베딩
    query_embedding = embed([query])[0]

    # 2. 벡터 검색
    results = _vector_store.search(query_embedding)

    # 3. 컨텍스트 구성
    context = "\n".join([r["document"] for r in results])

    # 4. LLM 호출 (Stub)
    response = generate(query, context)

    return {
        "query": query,
        "context": context,
        "response": response,
        "retrieved": results
    }
```

💡 학습 효율을 높이는 팁

1. 임베딩 모델 선택

```
# 경량 (90MB, 빠름)
'all-MiniLM-L6-v2'  # 기본 추천

# 중간 (420MB, 정확)
'all-mpnet-base-v2'

# 무거움 (1.3GB, 매우 정확)
'sentence-transformers/all-roberta-large-v1'
```

2. FAISS 인덱스 종류

```
# Flat (정확, 느림)
IndexFlatL2

# IVF (빠름, 근사)
IndexIVFFlat

# HNSW (매우 빠름, 근사)
IndexHNSWFlat
```

Day 5는 Flat 권장 (정확성)

3. 유사도 확인

```
# 검색 결과 거리 출력
for result in results:
    print(f"Distance: {result['distance']:.2f}")
    print(f"Doc: {result['document'][:50]}")
    print()

# 기준
# < 1.0: 매우 유사
# 1.0-2.0: 유사
# > 2.0: 다름
```

🔧 직접 해보기: 실습

실습 1: TOP_K 변경

config.py:

```
TOP_K = 5  # 3 → 5로 변경
```

결과: 더 많은 문서가 검색됨!

실습 2: 다른 임베딩 모델

config.py:

```
MODEL_NAME = "all-mpnet-base-v2"  # 더 정확한 모델
```

재시작 후 테스트: 검색 품질 비교!

실습 3: 유사도 시각화

test_similarity.py:

```
from embedding import embed

texts = [
    "강아지 키우기",
    "반려견 사육",
    "자동차 운전",
    "개 훈련법"
]

embeddings = embed(texts)

# 코사인 유사도
from sklearn.metrics.pairwise import cosine_similarity
sim_matrix = cosine_similarity(embeddings)

print("유사도 매트릭스:")
for i, text1 in enumerate(texts):
    for j, text2 in enumerate(texts):
        print(f"{text1} vs {text2}: {sim_matrix[i][j]:.2f}")
```

☑ Day 5 완료 기준

다음을 이해하고 실습했다면 완료입니다:

이해

- ✔ 왜 임베딩이 키워드 검색보다 좋은지
- ✔ Top-K 검색 방식
- ✔ Ingest 과정의 역할
- ✔ 검색과 생성의 분리 이유

코드

- ✔ embedding.py의 역할
- ✔ vectorstore.py의 작동 방식
- ✔ ingest.py의 실행 시점
- ✔ pipeline.py의 RAG 흐름

실습

- ✔ 패키지 설치 성공
- ✔ 서버 실행 및 Ingest 완료
- ✔ Swagger UI 테스트
- ✔ TOP_K 변경 후 비교
- ✔ 검색 결과 거리 확인

➡ SOON 다음 단계 (Day 6)

Day 6에서는:

Reranker & Hybrid Search:

```
# 1. Vector Search (의미 기반)
vector_results = vector_store.search(query, k=10)

# 2. BM25 Search (키워드 기반)
bm25_results = bm25.search(query, k=10)

# 3. Hybrid (결합)
combined = combine(vector_results, bm25_results)

# 4. Rerank (재정렬)
final = reranker.rank(combined, query)[:3]
```

검색 품질 대폭 향상!

💬 학습 후 자가 점검

다음 질문에 답할 수 있나요?

1. **임베딩이 키워드 검색보다 좋은 이유는?**
    - 의미 기반 검색, 동의어 인식, 문맥 이해
2. **FAISS IndexFlatL2의 L2는?**
    - L2 거리 (유클리드 거리) 기반 유사도
3. **Ingest는 언제 실행되나?**
    - 서버 시작 시 1회 (문서 → 임베딩 → 저장)

4. **TOP_K=3의 의미는?**
    - 가장 유사한 상위 3개 문서 검색
5. **검색과 생성을 분리하는 이유는?**
    - 독립적 최적화, 교체 용이, 테스트 명확

모두 답할 수 있다면 Day 5 완료! 🎉

📜 라이선스

⭐ **이 프로젝트가 도움이 되었다면 Star를 눌러주세요!**