

DAY 2 / CLASS 4

RAG (검색 증강 생성) 기반 AI 매뉴얼 시스템

Vector DB와 LangChain으로 구축하는 지능형 유지보수 도우미



TIME

14:00 ~ 15:30 (90min)



TOPIC

RAG, Vector DB, LangChain

왜 RAG 인가?

LLM의 한계와 검색 증강 생성의 필요성

! LLM PAIN POINTS



최신 정보 부재 (Cutoff Date)

"2024년 1월 이후의 데이터는 모릅니다."
학습 시점 이후의 변화에 대응 불가.



사내 데이터 접근 불가

"우리 공장의 보안 매뉴얼은 모릅니다."
외부에 공개되지 않은 내부 지식 부재.



환각 (Hallucination)

"모르면 모른다고 하지 않습니다."
그럴듯한 거짓말을 생성하여 신뢰도 하락.

💡 RAG SOLUTION



"Open Book Test"

AI가 기억에만 의존하지 않고,
참고서(매뉴얼)를 펼쳐보고 답안을 작성합니다.

VS

1

Retrieval (검색)

질문과 관련된 문서 조각을 Vector DB에서 찾음



2

Augmentation (증강)

찾은 정보를 프롬프트에 참고 자료로 끼워 넣음



3

Generation (생성)

참고 자료를 바탕으로 정확한 답변 생성



Vector DB와 임베딩 기초

Keyword Search vs Semantic Search



키워드 검색 (Keyword)

Exact Match (Ctrl+F)

사용자가 입력한 단어와 정확히 일치하는 텍스트만 찾아내는 전통적 방식.

- ✗ 단어 불일치 시 검색 실패 (예: '고장' ↔ '오작동')
- ✗ 문맥(Context)을 이해하지 못함
- ✓ 단순하고 명확한 ID/코드 검색에 유리



시맨틱 검색 (Semantic)

Vector Search (Embedding)

단어의 의미와 맥락을 수학적 거리로 계산하여 유사한 문서를 검색.

- ✓ 임베딩(Embedding): 텍스트를 숫자 벡터로 변환
- ✓ Vector DB: FAISS, Chroma 등 초고속 검색
- ✓ "배터리"를 검색해도 "전원" 관련 문서를 찾음

❶ 핵심 차이

기준 검색 (LEXICAL)

단어 일치 (Word)

글자가 같아야 함

vs

벡터 검색 (SEMANTIC)

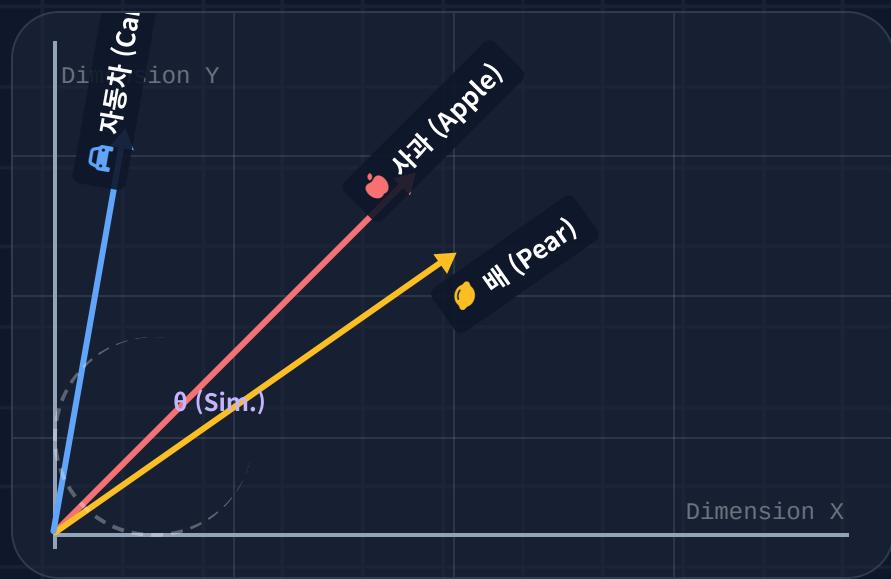
의미 유사 (Meaning)

뜻이 통하면 찾음

"임베딩이 텍스트를 숫자로 바꿔야 컴퓨터가
맥락을 이해합니다."

임베딩(Embedding) 원리 심화

| 컴퓨터는 어떻게 단어의 '의미'를 숫자로 이해하는가?



COSINE SIMILARITY FORMULA

$$\text{Similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

두 벡터 사이의 각도가 좁을수록(0°) 유사도는 1에 가깝다.



01

Tokenization (토큰화)

문장을 컴퓨터가 처리할 수 있는 최소 단위로 자르는 과정.

Ex) "배터리 교체" → ["배터리", "교체"]



02

Vectorization (좌표 변환)

각 토큰을 n차원(보통 1536차원)의 실수 리스트로 변환.

Ex) [0.12, -0.98, 0.05, ...]



03

Word2Vec vs Transformer

Word2Vec: 고정된 의미 (배 = 배)

Transformer: 문맥 기반 (먹는 배 vs 타는 배 구분)



04

Similarity (유사도 계산)

벡터 공간상에서 두 점 사이의 거리를 계산.

"사과"와 "배"는 가깝고, "자동차"는 멀게 배치됨.

시맨틱 검색 메커니즘

🔍 AI가 질문의 의도를 이해하고 문서를 찾아오는 5단계 프로세스



Performance Tuning Tips

검색 품질을 결정하는 핵심 변수: 차원 수(1536 vs 768), Index 타입(HNSW vs IVFFlat), Top-K(3~5개)



Optimization

> 환경 설정 및 매뉴얼 생성

RAG 구축을 위한 필수 라이브러리 설치 및 가상 데이터 준비



01_setup_and_data.py

```
# 1. RAG 구축을 위한 필수 라이브러리 설치
!pip install -q langchain langchain_community langchain_openai faiss-cpu tiktoken

import os
from langchain_openai import ChatOpenAI, OpenAIEMBEDDINGS

# API Key 설정 (실습 환경에 맞춰 설정)
# os.environ["OPENAI_API_KEY"] = "sk-..."

# 2. 가상의 제조 장비 매뉴얼 데이터 생성
robot_manual = """
[RB-X7 산업용 로봇 팔 유지보수 및 에러 대응 매뉴얼]
제 1장. 안전 수칙
- 로봇 반경 2m 이내 접근 시 반드시 티칭 펜던트의 비상 정지 버튼을 소지할 것.
- 유지보수 작업 전 메인 브레이커를 내리고 LOTO(Lock-Out, Tag-Out)를 실시할 것.
제 2장. 주요 에러 코드 및 조치
1. Error 4001 (Servo Motor Overheat)
- 원인: 2축 또는 3축 관절의 부하가 80%를 초과하여 지속됨. 또는 쿨링팬 고장.
- 조치:
  1) 로봇을 정지시키고 30분간 자연 냉각.
```

Key Libraries

- ✓ **langchain**: LLM 애플리케이션 프레임워크
- ✓ **faiss-cpu**: Meta에서 만든 고속 Vector DB 라이브러리
- ✓ **tiktoken**: OpenAI 모델용 토큰 계산기

Data Scenario

실습 편의를 위해 **로봇 팔 유지보수 매뉴얼**을 텍스트 변수로 직접 정의하고 파일로 저장합니다.

```
$ ls -l
-rw-r--r-- 1 user group 1024 Oct 25 14:00
robot_manual.txt
```

Step 2 문서 분할 (Chunking)

LLM이 소화할 수 있는 최적의 단위로 자르기



Why Chunking? (필요성)

1. LLM 용량 제한: 한 번에 책 한 권을 다 넣을 수 없음 (Token Limit)
2. 검색 정확도: 핀포인트로 정확한 문단을 찾아야 답변 품질 향상



Key Concepts (핵심 변수)

Chunk Size: 하나의 조각 크기 (문자 수)

Overlap: 앞뒤 조각 간 겹치는 구간 (맥락 유지)

◎ Best Practice Strategy

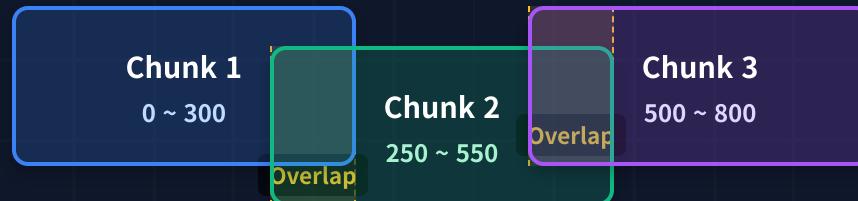
Size: 300~500 chars

Overlap: 30~80 chars

단순히 글자 수로 자르기보다, 문단(\n\n)이나 헤더(#) 기준으로 자르는 것이 의미 보존에 유리합니다.

Visualization

Original Long Document (Context)



□ Context A □ Context B □ Overlap Area

%

문서 분할 (Chunking) 코드

LLM이 이해하기 좋은 크기로 문서를 자르는 전처리 과정



02_chunking.py

```
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

# 1. 문서 로드 (앞서 저장한 매뉴얼 파일)
loader = TextLoader("robot_manual.txt")
documents = loader.load()

# 2. 문서 분할 (Chunking) 설정
# chunk_size: 한 조각의 최대 크기 (300자)
# chunk_overlap: 문맥 끊김 방지를 위한 중복 구간 (50자)
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=50
)

# 실제 분할 수행
splits = text_splitter.split_documents(documents)

print(f"== 문서 분할 결과: 총 {len(splits)}개의 조각(Chunk) 생성 ==")
```



Key Parameters

chunk_size=300 : 한 문단 정도의 크기

overlap=50 : 이전 문장의 끝부분을 포함

> Output Console

\$ python 02_chunking.py

== 문서 분할 결과: 총 3개의 조각(Chunk) 생성 ==

[Chunk 1 미리보기]

[RB-X7 산업용 로봇 팔 유지보수 및 에러 대응 매뉴얼]

제 1장. 안전 수칙

- 로봇 반경 2m 이내 접근 시 반드시 티칭 펜던트의 비상 정지 버튼을 소지할 것.
- 유지보수 작업 전 메인 브레이커를 내리고 LOTO(Lock-Out, Tag-Out)를 실시할 것.

임베딩 및 Vector DB 구축

텍스트를 벡터(숫자)로 변환하여 검색 엔진에 저장하기

3 03_embedding_vectorstore.py

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEMBEDDINGS

# 1. 임베딩 모델 준비 (문자를 숫자로 바꿔주는 번역기)
# cost-effective model: text-embedding-3-small
embeddings = OpenAIEMBEDDINGS(model="text-embedding-3-small")

# 2. 벡터 DB 생성 및 문서 저장
# 이 한 줄이 실행되면 모든 텍스트가 숫자로 변환되어 인덱싱됩니다.
# splits: 앞서 분할한 문서 조각들
vectorstore = FAISS.from_documents(
    documents=splits,
    embedding=embeddings
)

print("✅ Vector DB 구축 완료! (이제 시맨틱 검색이 가능합니다)")
```



INPUT DATA

Document Splits



MODEL API

OpenAI Ada-002 / 3-small



STORAGE

FAISS Vector Store



In-Memory Index

FAISS는 데이터를 RAM에 적재하여 밀리초(ms) 단위의 초고속 유사도 검색을 지원합니다.

🔍 검색(Retrieval) 성능 테스트

사용자 질문과 가장 의미가 가까운 문서 조각(Chunk)을 찾아오는지 확인

● ● ● 04_retrieval_test.py

```
# 1. Retriever(검색기) 생성
# search_type="similarity": 코사인 유사도 기반 검색
# k=2: 가장 유사도가 높은 상위 2개 문서만 반환
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 2}
)

# 2. 테스트 질문 (키워드가 아닌 문장형 질문)
query = "엔코더 배터리가 부족할 때 어떻게 해야 해?"

# 3. 검색 수행
retrieved_docs = retriever.invoke(query)

# 결과 출력
print(f"User Query: {query}\n")
print("== [검색된 관련 문서 조각] ==")
for i, doc in enumerate(retrieved_docs):
    print(f"[문서 {i+1}]: {doc.page_content}...")
```

>_ Console Output

User Query: 엔코더 배터리가 부족할 때 어떻게 해야 해?

== [검색된 관련 문서 조각] ==

[문서 1] Similarity: 0.89

2. Error 5005 (Encoder Battery Low)

- 원인: 위치 정보를 기억하는 엔코더의 백업 배터리 전압 저하.
- 조치: 전월이 켜진 상태에서 3.6V 리튬 배터리로 교체할 것.

[문서 2] Similarity: 0.72

제 1장. 안전 수칙

- 유지보수 작업 전 메인 브레이커를 내리고 LOTO를 실시할 것.

✅ Semantic Match Success!

💡 Observation

사용자가 정확한 에러 코드(Error 5005)를 말하지 않아도, '배터리 부족'이라는 의미(Semantic)를 파악하여 정확한 매뉴얼 페이지를 찾아냈습니다.

📝 RAG 프롬프트 설계

할루시네이션을 방지하고 정확한 답변을 유도하는 프롬프트 엔지니어링



05_rag_prompt.py

```
from langchain_core.prompts import ChatPromptTemplate

# 1. RAG 전용 시스템 프롬프트 정의
rag_system_prompt = """
당신은 로봇 유지보수 지원 AI입니다.
아래의 [참고 문서]를 바탕으로 사용자의 질문에 답변하세요.
규칙:
1. 문서에 없는 내용은 절대 지어내지 말고 '매뉴얼에 내용이 없습니다'라고 답하세요.
2. 답변은 전문가스러운 톤으로 간결하게 작성하세요.
3. 답변 끝에 근거가 되는 챕터나 에러 코드를 명시하세요.

[참고 문서]
{context}
"""

# 2. ChatPromptTemplate 생성
# {context}: 검색된 문서 조각들이 들어갈 자리
# {input}: 사용자의 질문이 들어갈 자리
prompt = ChatPromptTemplate.from_messages([
    ("system", rag_system_prompt),
    ("human", "{input}"),
])
```

☑ Prompt Checklist

- 💡 **Grounding (근거 기반):**
반드시 제공된 `{context}` 내에서만 답변하도록 강제하여 할루시네이션(거짓말)을 차단합니다.
- 🚫 **Fallback (예외 처리):**
모르는 내용은 "모른다"고 답하게 하여 잘못된 유지보수 정보 전달을 방지합니다.
- 👤 **Persona (페르소나):**
"로봇 유지보수 지원 AI"라는 역할을 부여하여 답변의 톤앤매너를 전문적으로 유지합니다.

💡 Structure Tip

LangChain의 체인(Chain)이 실행될 때, Retriever가 찾아온 문서 조각들이 자동으로 `{context}` 변수 자리에 주입(Injection)됩니다.

▶ 최종 RAG 체인 실행

검색(Retrieval)과 생성(Generation)을 결합하여 디지털 트윈 에러 해결 솔루션 도출



05_rag_chain_execution.py

```
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain

# 1. 문서 결합 체인 생성 (검색된 문서를 프롬프트에 주입)
# llm과 prompt는 이전 단계에서 정의됨
question_answer_chain = create_stuff_documents_chain(llm, prompt)

# 2. RAG 체인 생성 (Retriever + QA Chain)
rag_chain = create_retrieval_chain(retriever, question_answer_chain)

# 3. 상황 발생: 디지털 트윈에서 에러 감지 (시뮬레이션)
dt_error_msg = "현재 로봇 컨트롤러에 Error 4001 알람이 발생했습니다. 원인과 조치 방법을 알려주세요."

print(f"User Question: {dt_error_msg}\n")
print("== [AI Manual Bot Answer] ==")

# 4. 체인 실행 (Invoke)
response = rag_chain.invoke({"input": dt_error_msg})
print(response["answer"])
```

⚠ Digital Twin Alert



Error 4001 Detected

Servo Motor Overheat

Real-time

🤖 AI Manual Bot Answer

Generated

Based on [RB-X7 Maintenance Manual]:

🔍 추정 원인

- 2축/3축 관절 부하 80% 초과 지속
- 제어기 쿨링팬 고장

🔧 조치 방법

1. 로봇 정지 후 30분간 자연 냉각
2. 쿨링팬 동작 확인 및 필터 청소
3. 감속기 윤활유(Grease) 오염 점검 및 교체

🏁 2일간의 여정 완주!

데이터에서 지능까지: 차세대 인지형 제조 시스템 구축 완료



Course Achievements

- ✓ ① 데이터 수집 (센서)
다양한 센서 데이터를 표준 프로토콜로 수집하여 파이프라인 구축 완료
- ✓ ② 이상 탐지 (신경)
노이즈 섞인 시계열 데이터를 전처리하고 임계값 기반 이상 징후 포착
- ✓ ③ Agent 분석 (두뇌)
LangChain Agent를 활용해 데이터의 패턴을 스스로 분석하고 인사이트 도출
- ✓ ④ RAG 기억 (지식)
Vector DB에 매뉴얼을 학습시켜 전문 지식에 기반한 해결책 제시 시스템 구현



이제 데이터 + 두뇌 + 기억을 연결하면 차세대 인지형 제조 시스템이 완성됩니다!



Mission Complete

Q & A



오늘 학습한 RAG와 Vector DB에 대해 무엇이든 물어보세요.

SUGGESTED DISCUSSION TOPICS



RAG 실제 적용 사례



Vector DB 선택 가이드



LangChain 고급 패턴

Day 2 Completed