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

---

VectorStoreRetrieverMemory

- 대화 내용을 벡터(숫자 배열)로 변환 하여 저장하고, **필요할 때 검색 해서 불러오는 메모리**
- 즉, 벡터 스토어 에 메모리 저장 → 호출될 때마다 가장 **눈에 띄는 상위 K개의 문서 를 쿼리**

- **다른 메모리 클래스와의 차이점** : 대화 내용의 **순서를 명시적으로 추적하지 않는다는 점**

- **역할**:
  - 매우 방대한 양의 정보를 저장 하고, 사용자가 질문하는 내용과 **가장 유사한 정보를 찾아냄**
  - **ㄴ 원하는 정보를 검색창에 입력하면 수많은 자료 중에서 찾아주는 도서관 검색 시스템**

```
# 환경변수 처리 및 클라이언트 생성
from langsmith import Client
from langchain.prompts import PromptTemplate
from langchain.prompts import ChatPromptTemplate
from dotenv import load_dotenv

import os
import json

# 클라이언트 생성
api_key = os.getenv("LANGSMITH_API_KEY")
client = Client(api_key=api_key)


# LangSmith 추적 설정하기 (https:smith.langchin.com)
# LangSmith 추적을 위한 라이브러리 임포트
from langsmith import traceable

# 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_A
org = "설정됨" if os.getenv('LANGCHAIN_ORGANIZATION') else "설

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_
    print(f"✅ LangSmith 추적 활성화됨 (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 확
    if langchain_tracing_v2 != "true":
        print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

"@traceable" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

- 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.
```

```
import os
from dotenv import load_dotenv
import openai

# .env 파일에서 환경변수 불러오기
load_dotenv()

# 환경변수에서 API 키 가져오기
api_key = os.getenv("OPENAI_API_KEY")

# OpenAI API 키 설정
openai.api_key = api_key

# OpenAI를 불러오기
# ✅ 디버깅 함수: API 키가 잘 불러와졌는지 확인
def debug_api_key():
    if api_key is None:
        print("❌ API 키를 불러오지 못했습니다. .env 파일과 변수명을 확인하세요.")
    elif api_key.startswith("sk-") and len(api_key) > 20:
        print("✅ API 키를 성공적으로 불러왔습니다.")
    else:
        print("⚠️ API 키 형식이 올바르지 않은 것 같습니다. 값을 확인하세요.")

# 디버깅 함수 실행
debug_api_key()
```

- 셀 출력

```
✅ API 키를 성공적으로 불러왔습니다.
```

```
import os
from dotenv import load_dotenv
from openai import OpenAI
from langchain_openai import ChatOpenAI

# .env 파일에서 환경변수 불러오기
load_dotenv()

# 환경변수에서 API 키 가져오기
api_key = os.getenv("OPENAI_API_KEY")

# OpenAI API 키 설정
openai.api_key = api_key
```

- 벡터 스토어 초기화하기
- 사전 터미널에 설치 필요한 벡터스토어 관련 내용
  - CPU ver.

```
pip install faiss-cpu
```

*

- GPU ver.

```
pip install faiss-gpu
```

```
import faiss                                                    # Meta AI Research에서 개발한 라이브러리
from langchain_openai import OpenAIEmbeddings
```

```
from langchain.docstore import InMemoryDocstore
from langchain.vectorstores import FAISS


# 임베딩 모델 정의하기
embeddings_model = OpenAIEmbeddings(
    model="text-embedding-3-small",                          # 실제 임베딩 모델명 넣기
    api_key=api_key                                          # 임베딩 모델용 api_key를 환경변수 처리 후 넣기
)

# Vector Store 초기화하기
embedding_size = 1536
index = faiss.IndexFlatL2(embedding_size)
vectorstore = FAISS(                                         # FAISS 형태 최신 코드 형태로 바꿔서 써야 오류 생기지 않음
    embedding_function=embeddings_model,
    index=index,
    docstore=InMemoryDocstore({}),
    index_to_docstore_id={})
```

- **K = 1**
    - 실제 사용에서는 k 를 더 높은 값으로 설정

```
from langchain.memory import VectorStoreRetrieverMemory

# 벡터 조회가 여전히 의미적으로 관련성 있는 정보를 반환한다는 것을 보여주기 위해서 설정함
retriever = vectorstore.as_retriever(search_kwargs={"k": 1})
memory = VectorStoreRetrieverMemory(retriever=retriever)
```

- 셀 출력

```
/var/folders/h3/l7wnkv352kqftv0t8ctl2ld40000gn/T/ipykernel_15381/1689916482.py:5: LangChainDeprecationWarning: Pl
memory = VectorStoreRetrieverMemory(retriever=retriever)
```

- 셀 출력 해석
    - 경고 메시지

```
LangChainDeprecationWarning: Please see the migration guide...
```

        - `DeprecationWarning` = **이 기능은 곧 없어질 예정이니, 새 방법을 쓰세요** 라는 의미
        - `VectorStoreRetrieverMemory` 라는 기능이 앞으로 사라질 예정 이라, 지금은 쓸 수 있지만 미래 버전에서는 없어질 수 있다 는 의미
        - `LangChain` 의 새로운 가이드 사이드 안내

- 해당 기능이 사라지는 이유
    - `LangChain` 의 버전이 올라가면서 **메모리 시스템 구조를 새롭게 통합 중**
    - 이전 방식(`VectorStoreRetrieverMemory`): 유지•관리 어려움 + 새로운 기능들과의 호환성이 떨어짐
    - **더 유연하고 표준화된 메모리 구조로 교체하는 중**

- 최신 코드에서 쓰는 방법
    - 권장_1: `RunnableWithMessageHistory` + `VectorStoreRetriever` 조합
    - 권장_1의 예시

```
# 필요한 라이브러리 임포트
  from langchain_openai import ChatOpenAI, OpenAIEmbeddings
  from langchain_community.vectorstores import FAISS
  from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
  from langchain_core.runnables.history import RunnableWithMessageHistory
  from langchain_community.chat_message_histories import ChatMessageHistory
  from langchain_core.chat_history import BaseChatMessageHistory

  # 1. 벡터스토어 + 리트리버 준비
  embeddings = OpenAIEmbeddings()
  # 예시 문서
  docs = [
      {"page_content": "Python is a programming language."},
      {"page_content": "LangChain helps build LLM-powered applications."},
      {"page_content": "FAISS is a vector database for similarity search."}
  ]
  vectorstore = FAISS.from_documents(docs, embedding=embeddings)
  retriever = vectorstore.as_retriever()

  # 2. 프롬프트 템플릿 (대화 기록 포함)
  prompt = ChatPromptTemplate.from_messages([
      ("system", "당신은 친절한 AI 어시스턴트입니다. 질문에 답하세요."),
      MessagesPlaceholder(variable_name="history"),  # 대화 기록 자리
      ("human", "{input}"),
  ])

  # 3. LLM 모델
  llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

  # 4. 리트리버와 LLM을 연결하는 Runnable
  #    (검색 결과를 context로 넣어주는 간단한 체인)
  def retrieval_chain(inputs):
      query = inputs["input"]
      docs = retriever.get_relevant_documents(query)
      context = "\n".join([d.page_content for d in docs])
      return prompt.format_messages(history=inputs["history"], input=f"{query}\n\n참고자료:\n{context}")

  # 5. 세션별 대화 기록 저장소
  store = {}
  def get_session_history(session_id: str) -> BaseChatMessageHistory:
      if session_id not in store:
          store[session_id] = ChatMessageHistory()
      return store[session_id]

  # 6. RunnableWithMessageHistory로 감싸기
  with_history = RunnableWithMessageHistory(
      runnable = retrieval_chain | llm,
      get_session_history = get_session_history,
      input_messages_key = "input",       # 사용자 입력 키
      history_messages_key = "history"    # 대화 기록 키
  )

  # 7. 실행 예시
  session_id = "user-123"

  response1 = with_history.invoke(
      {"input": "LangChain이 뭐야?"},
      config={"configurable": {"session_id": session_id}}
  )
  print(response1)

  response2 = with_history.invoke(
      {"input": "FAISS에 대해서도 알려줘"},
      config={"configurable": {"session_id": session_id}}
```

```
    )
    print(response2)
```

---

* * 권장_2: `ConversationBufferMemory` 또는 `ConversationBufferWindowMemory` 같은 새 메모리 클래스를 쓰는 걸 권장 * 권장_2의 예시
```python from langchain.memory import ConversationBufferMemory from langchain.chains import ConversationChain from langchain.chat_models import ChatOpenAI

```
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
llm = ChatOpenAI()
conversation = ConversationChain(llm=llm, memory=memory)

conversation.predict(input="안녕!")
```
```

```
# 임의의 대화 저장해보기

memory.save_context(
    inputs={
        "human": "안녕하세요, 오늘 면접에 참석해주셔서 감사합니다. 자기소개 부탁드립니다."
    },
    outputs={
        "ai": "안녕하세요. 저는 컴퓨터 과학을 전공한 신입 개발자입니다. 대학에서는 주로 자바와 파이썬을 사용했으며, 최근에는 웹 개발 프로젝트에 참여하여 실제 사
    },
)
memory.save_context(
    inputs={"human": "프로젝트에서 어떤 역할을 맡았나요?"},
    outputs={
        "ai": "제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을 담당했습
    },
)
memory.save_context(
    inputs={
        "human": "팀 프로젝트에서 어려움을 겪었던 경험이 있다면 어떻게 해결했나요?"
    },
    outputs={
        "ai": "프로젝트 초기에 의사소통 문제로 몇 가지 어려움이 있었습니다. 이를 해결하기 위해 저희 팀은 정기적인 미팅을 갖고 각자의 진행 상황을 공유했습니다. 또
    },
)
memory.save_context(
    inputs={"human": "개발자로서 자신의 강점은 무엇이라고 생각하나요?"},
    outputs={
        "ai": "제 강점은 빠른 학습 능력과 문제 해결 능력입니다. 새로운 기술이나 도구를 빠르게 습득할 수 있으며, 복잡한 문제에 직면했을 때 창의적인 해결책을 제시
    },
)
```

- 다음의 질문을 했을 때 Vector Store 로 부터 1개(k=1 이기 때문)의 가장 관련성 높은 대화를 반환

  ```
  질문: **"면접자 전공은 무엇인가요?"**
  ```

```
# 메모리에 질문을 통해 가장 연관성 높은 1개 대화를 추출

print(memory.load_memory_variables({"prompt": "면접자 전공은 무엇인가요?"})["history"])
```

- 셀 출력

  ```
  human: 안녕하세요, 오늘 면접에 참석해주셔서 감사합니다. 자기소개 부탁드립니다.
  ai: 안녕하세요. 저는 컴퓨터 과학을 전공한 신입 개발자입니다. 대학에서는 주로 자바와 파이썬을 사용했으며, 최근에는 웹 개발 프로젝트에 참여하여 실제 
  ```

- 이번에는 다른 질문을 통해 가장 연관성 높은 1개 대화를 추출합니다.

  ```
  질문: **"면접자가 프로젝트에서 맡은 역할은 무엇인가요?"**
  ```

```
print(
    memory.load_memory_variables(
        {"human": "면접자가 프로젝트에서 맡은 역할은 무엇인가요?"}
    )["history"]
)
```

- 셀 출력

```
human: 프로젝트에서 어떤 역할을 맡았나요?
ai: 제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을 담당했
```

업데이트된 방식 시도

- 교재 코드 오류 발생
  - 원인_1: 임베딩 모델 제대로 생성 X
    - `OpenAIEmbeddings()` 초기화 시 **필수 인자 누락**
      - `langchain_openai.OpenAIEmbeddings`는 기본값 없음 → `model`과 `API 키`를 지정해야 함
      - `gpt-4o-mini` = 텍스트 생성 모델 → 벡터스토어에 쓸 임베딩은 **별도의 임베딩 전용 모델 지정 필요**
        - `GPT-4o-mini`는 그대로 대화/생성에 쓰고, 검색용 벡터는 임베딩 모델로 생성하는 구조가 일반적
        - 임베딩 모델 예시: **`text-embedding-3-small`** 또는 **`text-embedding-3-large`**
  - 원인_2: **`FAISS` 최신 초기화 방식 변경**
    - 예전처럼 `FAISS(embeddings_model, index, ...)`로 직접 생성하는 방식은 **deprecated**
    - **최신** 버전에서는 `FAISS.from_texts()` 또는 `FAISS()`에 `embedding_function=` 키워드로 넘겨야 함

- 최신 방식으로 수정한 코드로 도전

```
import os
import time
import openai
from dotenv import load_dotenv

# .env 파일에서 API 키 불러오기
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

def safe_chat_request(prompt, model="gpt-4o-mini", max_retries=5):
    """429 에러 방지를 위한 안전 호출 함수"""
    retries = 0
    wait_time = 1  # 첫 대기 시간(초)

    while retries < max_retries:
        try:
            response = openai.ChatCompletion.create(
                model=model,
                messages=[{"role": "user", "content": prompt}],
                temperature=0.7
            )
            return response.choices[0].message["content"]

        except openai.error.RateLimitError:
            print(f"[경고] 요청이 너무 많습니다. {wait_time}초 후 재시도...")
            time.sleep(wait_time)
            wait_time *= 2  # 지수 백오프
            retries += 1

        except Exception as e:
            print(f"[에러] {e}")
            break

    return "요청 실패: 재시도 횟수를 초과했습니다."
```

```
# 사용 예시
if __name__ == "__main__":
    answer = safe_chat_request("안녕하세요, 오늘 날씨에 맞는 점심 메뉴 추천해줘")
    print(answer)
```

```
import os
import time
from openai import OpenAI
#from openai.error import RateLimitError  # 예외 경로 변경됨
from openai import RateLimitError
from dotenv import load_dotenv

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

def safe_chat_request(prompt, model="gpt-4o-mini", max_retries=5):
    retries = 0
    wait_time = 1

    while retries < max_retries:
        try:
            response = client.chat.completions.create(
                model=model,
                messages=[{"role": "user", "content": prompt}],
                temperature=0.7
            )
            return response.choices[0].message.content

        except RateLimitError:
            print(f"[경고] 요청이 너무 많습니다. {wait_time}초 후 재시도...")
            time.sleep(wait_time)
            wait_time *= 2
            retries += 1

        except Exception as e:
            print(f"[에러] {e}")
            break

    return "요청 실패: 재시도 횟수를 초과했습니다."

if __name__ == "__main__":
    answer = safe_chat_request("안녕하세요, 오늘 날씨에 맞는 점심 메뉴 추천해줘")
    print(answer)
```

```
안녕하세요! 오늘 날씨가 어떤지에 따라 추천해드릴 수 있는 점심 메뉴가 달라질 수 있어요. 날씨가 맑고 따뜻하다면 샐러드나 가벼운 파스타가 좋고, 비나 추운 날씨라면
```

- 셀 출력_1 (2.2s)

```
안녕하세요! 오늘 날씨에 따라 점심 메뉴를 추천해드릴게요. 날씨가 맑고 따뜻하다면 신선한 샐러드나 냉면이 좋고, 비가 오거나 쌀쌀하다면 따뜻한 국물 요리
```

- 셀 출력_2 (2.1s)
  - 모듈 임포트 중 과거 버전 → 최신 방식으로 변경 추가 후 다시 실행
    - 과거 임포트 방법 → 오류 발생 → 셀 출력_1 에서 제외하고 실행

      ```
      from openai.error import RateLimitError
      ```

    - 최신 임포트 방법 → 추가 → 셀 출력_2 에서 추가 및 실행

      ```
      from openai import RateLimitError
      ```

```
안녕하세요! 오늘 날씨가 어떤지에 따라 추천해드릴 수 있는 점심 메뉴가 달라질 수 있어요. 날씨가 맑고 따뜻하다면 샐러드나 가벼운 파스타가 좋고, ㅂ
```

📌 실행 구조

- 문서 임베딩 생성 → `OpenAI Embeddings API` 사용
- 벡터스토어에 저장 → 예: `FAISS`, `Chroma`, `Weaviate` 등
- 질의 시 → 질문을 임베딩으로 변환 → 벡터스토어에서 유사도 검색
- 검색 결과 + 사용자 질문을 GPT 모델에 전달 → 최종 답변 생성

```
import os
import time
from openai import OpenAI, RateLimitError
from dotenv import load_dotenv
import faiss
import numpy as np

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# 1. 안전 호출 함수 (429 방지)
def safe_chat_request(messages, model="gpt-4o-mini", max_retries=5):
    retries, wait_time = 0, 1
    while retries < max_retries:
        try:
            resp = client.chat.completions.create(
                model=model,
                messages=messages,
                temperature=0.7
            )
            return resp.choices[0].message.content
        except RateLimitError:
            print(f"[경고] 요청이 많습니다. {wait_time}초 후 재시도...")
            time.sleep(wait_time)
            wait_time *= 2
            retries += 1
        except Exception as e:
            print(f"[에러] {e}")
            break
    return None

# 2. 텍스트 → 벡터 변환
def get_embedding(text, model="text-embedding-3-small"):
    emb = client.embeddings.create(
        model=model,
        input=text
    )
    return np.array(emb.data[0].embedding, dtype="float32")

# 3. 샘플 문서 벡터스토어 구축
docs = [
    "파이썬은 데이터 분석과 AI 개발에 널리 사용되는 언어입니다.",
    "FAISS는 Facebook AI Research에서 만든 벡터 검색 라이브러리입니다.",
    "벡터스토어는 문서 검색과 추천 시스템에 활용됩니다."
]

# FAISS 초기화
dimension = len(get_embedding("test"))
index = faiss.IndexFlatL2(dimension)

# 문서 임베딩 저장
doc_embeddings = [get_embedding(doc) for doc in docs]
index.add(np.array(doc_embeddings))

# 4. 검색 + GPT 연결
def search_and_answer(query):
    query_vec = get_embedding(query)
    D, I = index.search(np.array([query_vec]), k=2)  # 상위 2개 검색
    retrieved_docs = [docs[i] for i in I[0]]

    # GPT에 전달할 메시지 구성
    messages = [
        {"role": "system", "content": "당신은 검색 기반 질문에 답하는 AI입니다."},
        {"role": "user", "content": f"다음 문서들을 참고해서 답변해 주세요:\n{retrieved_docs}\n\n질문: {query}"}
    ]
    return safe_chat_request(messages)

# 실행 예시
if __name__ == "__main__":
```

```
    answer = search_and_answer("벡터스토어가 뭐야?")
    print("답변:", answer)


import os
from dotenv import load_dotenv
import faiss
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_community.vectorstores import FAISS
from langchain.memory import VectorStoreRetrieverMemory

# ========================================
# 1. OpenAI API 키 설정
# ========================================
# 환경변수에 API 키 저장
# .env 파일에서 환경변수 불러오기
load_dotenv()

# 환경변수에서 API 키 가져오기
api_key = os.getenv("OPENAI_API_KEY")

# OpenAI API 키 설정
openai.api_key = api_key

# ========================================
# 2. 임베딩 모델 정의
# ========================================
# gpt-4o-mini는 텍스트 생성용, 벡터 검색에는 임베딩 전용 모델 사용
embeddings_model = OpenAIEmbeddings(
    model="text-embedding-3-small",                # 또는 "text-embedding-3-large"
    api_key=api_key
)

# ========================================
# 3. 빈 FAISS 인덱스 초기화
# ========================================
embedding_size = len(embeddings_model.embed_query("테스트"))      # 모델 차원 자동 계산
index = faiss.IndexFlatL2(embedding_size)                         # L2 거리 기반 인덱스
vectorstore = FAISS(                                              # FAISS 최신 방식으로 변수 지정
    embedding_function=embeddings_model,
    index=index,
    docstore=InMemoryDocstore({}),
    index_to_docstore_id={}
)

# ========================================
# 4. RetrieverMemory 연결
# ========================================
retriever = vectorstore.as_retriever(search_kwargs={"k": 1})
memory = VectorStoreRetrieverMemory(retriever=retriever)

# ========================================
# 5. 대화 내용 저장
# ========================================
memory.save_context(
    inputs={"human": "안녕하세요, 오늘 면접에 참석해주셔서 감사합니다. 자기소개 부탁드립니다."},
    outputs={"ai": "안녕하세요. 저는 컴퓨터 과학을 전공한 신입 개발자입니다. 대학에서는 주로 자바와 파이썬을 사용했으며, 최근에는 웹 개발 프로젝트에 참여하여
)
memory.save_context(
    inputs={"human": "프로젝트에서 어떤 역할을 맡았나요?"},
    outputs={"ai": "제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을
)
memory.save_context(
    inputs={"human": "팀 프로젝트에서 어려움을 겪었던 경험이 있다면 어떻게 해결했나요?"},
    outputs={"ai": "프로젝트 초기에 의사소통 문제로 몇 가지 어려움이 있었습니다. 이를 해결하기 위해 저희 팀은 정기적인 미팅을 갖고 각자의 진행 상황을 공유했습
)
memory.save_context(
    inputs={"human": "개발자로서 자신의 강점은 무엇이라고 생각하나요?"},
    outputs={"ai": "제 강점은 빠른 학습 능력과 문제 해결 능력입니다. 새로운 기술이나 도구를 빠르게 습득할 수 있으며, 복잡한 문제에 직면했을 때 창의적인 해결책
)

# ========================================
# 6. 메모리 검색 테스트
# ========================================
print(memory.load_memory_variables({"prompt": "면접자 전공은 무엇인가요?"})["history"])
print(memory.load_memory_variables({"human": "면접자가 프로젝트에서 맡은 역할은 무엇인가요?"})["history"])
```

```
# ========================================
# 7. gpt-4o-mini와 결합 예시
# ========================================
llm = ChatOpenAI(model="gpt-4o-mini", api_key=os.environ["OPENAI_API_KEY"])

question = "이 면접자의 강점을 한 문장으로 요약해줘."
context = memory.load_memory_variables({"human": question})["history"]

response = llm.invoke(f"다음 대화 기록을 참고해서 답변해줘:\n{context}\n\n질문: {question}")
print(response.content)
```

- 셀 출력 오류: 크레딧 소진으로 현재 해결 어려움 → 다른 임베딩 모델 사용해보기로 함

- **임베딩 모델 교체 필요: `OpenAI 모델` → `all-MiniLM-L6-v2`**
- **`sentence-transformers/all-MiniLM-L6-v2`**
  - 역할: 문장이나 짧은 문단을 384차원 벡터로 변환 → **의미 기반 검색, 유사도 계산** 등에 사용
  - 기반: BERT 계열의 MiniLM 아키텍처를 1B(10억) 문장 쌍 데이터로 파인튜닝
  - 언어: 주로 영어 에 최적화, 하지만 한국어도 어느 정도 처리 가능 (다만 정확도는 다국어 모델보다 낮음)
  - 속도: CPU에서도 빠르게 동작, GPU 사용 시 더 빠름
- 용량
  - 모델 파일 크기: 약 90MB (PyTorch weights 기준)
  - 설치 후 캐시 포함: 약 100~120MB 정도 차지
  - 메모리 사용량: CPU 에서 로드 시 약 400~500MB RAM 사용
- 저장 위치: 전약 or 해당 프로젝트

- 임베딩 모델을 활용한 전체 코드

```
# test_1

from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

# 로컬 임베딩 모델 로드
embed_model = SentenceTransformer('all-MiniLM-L6-v2')

docs = [
    "파이썬은 데이터 분석과 AI 개발에 널리 사용되는 언어입니다.",
    "FAISS는 Facebook AI Research에서 만든 벡터 검색 라이브러리입니다.",
    "벡터스토어는 문서 검색과 추천 시스템에 활용됩니다."
]

# 문서 임베딩
doc_embeddings = embed_model.encode(docs, convert_to_numpy=True)

# FAISS 인덱스 생성
dimension = doc_embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(doc_embeddings)

# 검색 함수
def search(query, k=2):
    query_vec = embed_model.encode([query], convert_to_numpy=True)
    D, I = index.search(query_vec, k)
    return [docs[i] for i in I[0]]

print(search("벡터스토어 설명해줘"))
```

- 셀 출력 (23.4s)

```
/Users/jay/.pyenv/versions/lc_env/lib/python3.13/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found.
from .autonotebook import tqdm as notebook_tqdm
```

```
['벡터스토어는 문서 검색과 추천 시스템에 활용됩니다.', '파이썬은 데이터 분석과 AI 개발에 널리 사용되는 언어입니다.']
```

- 셀 출력 경고 메시지
  - 분석
    - /Users/jay/.pyenv/versions/lc_env/lib/python3.13/site-packages/tqdm/auto.py:21
      - 경고가 발생한 파일 경로와 줄 번호
      - tqdm/auto.py → tqdm의 자동 환경 감지 모드 파일
      - :21 → 21번째 줄에서 경고를 발생시켰다는 뜻
    - `TqdmWarning`
      - `tqdm` 라이브러리가 정의한 경고 클래스 이름
      - `Warning` 이므로 코드 실행은 계속되지만, 기능 일부가 제한될 수 있다는 신호
    - `IProgress not found.`
      - `IProgress` = **`Jupyter Notebook/Lab` 에서 진행바를 표시하는 위젯 클래스**
      - `ipywidgets` 패키지 안에 포함되어 있음
      - 현재 환경에서 IProgress를 찾지 못했다는 뜻 → 즉, ipywidgets가 없거나 버전이 맞지 않음
    - `Please update jupyter and ipywidgets.`
      - 해결 방법 안내_1: **`jupyter` (노트북 실행 환경)**
      - 해결 방법 안내_2: **`ipywidgets` (진행바, 슬라이더, 버튼 같은 UI 위젯 제공)**
      - 이 둘을 설치하거나 최신 버전으로 업데이트하라는 의미
    - `See https://ipywidgets.readthedocs.io/en/stable/user_install.html`
    - = 공식가이드 참고하라는 의미
    - `from .autonotebook import tqdm as notebook_tqdm`
      - `tqdm`이 `Jupyter` 환경이면 `notebook`용 진행바를, 터미널 환경이면 `CLI`용 진행바를 **자동으로 선택하려는 코드**
      - 여기서 `notebook`용 진행바를 쓰려고 하다가 **IProgress가 없어서 경고 발생**

```
# test_2
# vscode 터미널: `pip install ipywidgets` 설치 후

from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

# 로컬 임베딩 모델 로드
embed_model = SentenceTransformer('all-MiniLM-L6-v2')

docs = [
    "파이썬은 데이터 분석과 AI 개발에 널리 사용되는 언어입니다.",
    "FAISS는 Facebook AI Research에서 만든 벡터 검색 라이브러리입니다.",
    "벡터스토어는 문서 검색과 추천 시스템에 활용됩니다."
]

# 문서 임베딩
doc_embeddings = embed_model.encode(docs, convert_to_numpy=True)

# FAISS 인덱스 생성
dimension = doc_embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(doc_embeddings)

# 검색 함수
def search(query, k=2):
    query_vec = embed_model.encode([query], convert_to_numpy=True)
    D, I = index.search(query_vec, k)
    return [docs[i] for i in I[0]]

print(search("벡터스토어 설명해줘"))
```

- 셀 출력 (3.5s)

```
['벡터스토어는 문서 검색과 추천 시스템에 활용됩니다.', '파이썬은 데이터 분석과 AI 개발에 널리 사용되는 언어입니다.']
```

- 교재 속 내용으로 코드 수정해보기

```
import os
import faiss
from sentence_transformers import SentenceTransformer
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_community.vectorstores import FAISS
from langchain.memory import VectorStoreRetrieverMemory
from langchain_openai import ChatOpenAI                                    # GPT 호출용 (원하면 제거 가능)

# ========================================
# 1. 로컬 임베딩 모델 로드
# ========================================
# 'cache_folder'를 지정하면 모델이 프로젝트 폴더 안에 저장됨
# 처음 실행 시 Hugging Face Hub에서 다운로드 후 캐시에 저장
embed_model = SentenceTransformer(
    'sentence-transformers/all-MiniLM-L6-v2',
    cache_folder="./models"                                                # 프로젝트 루트의 models 폴더에 저장
)

# ========================================
# 2. 빈 FAISS 인덱스 초기화
# ========================================
# 모델의 임베딩 차원 수를 자동 계산
embedding_size = embed_model.get_sentence_embedding_dimension()

# L2 거리 기반 인덱스 생성
index = faiss.IndexFlatL2(embedding_size)

# LangChain의 FAISS 래퍼로 감싸기
vectorstore = FAISS(
    embedding_function=embed_model.encode,                                 # 로컬 임베딩 함수 사용
    index=index,
    docstore=InMemoryDocstore({}),
    index_to_docstore_id={}
)

# ========================================
# 3. RetrieverMemory 연결
# ========================================
retriever = vectorstore.as_retriever(search_kwargs={"k": 1})               # k=1 → 가장 유사한 1개만 검색해보기
memory = VectorStoreRetrieverMemory(retriever=retriever)

# ========================================
# 4. 대화 내용 저장
# ========================================
memory.save_context(
    inputs={"human": "안녕하세요, 오늘 면접에 참석해주셔서 감사합니다. 자기소개 부탁드립니다."},
    outputs={"ai": "안녕하세요. 저는 컴퓨터 과학을 전공한 신입 개발자입니다. 대학에서는 주로 자바와 파이썬을 사용했으며, 최근에는 웹 개발 프로젝트에 참여하여
)
memory.save_context(
    inputs={"human": "프로젝트에서 어떤 역할을 맡았나요?"},
    outputs={"ai": "제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을
)
memory.save_context(
    inputs={"human": "팀 프로젝트에서 어려움을 겪었던 경험이 있다면 어떻게 해결했나요?"},
    outputs={"ai": "프로젝트 초기에 의사소통 문제로 몇 가지 어려움이 있었습니다. 이를 해결하기 위해 저희 팀은 정기적인 미팅을 갖고 각자의 진행 상황을 공유했습
)
memory.save_context(
    inputs={"human": "개발자로서 자신의 강점은 무엇이라고 생각하나요?"},
    outputs={"ai": "제 강점은 빠른 학습 능력과 문제 해결 능력입니다. 새로운 기술이나 도구를 빠르게 습득할 수 있으며, 복잡한 문제에 직면했을 때 창의적인 해결책
)
```

```
`embedding_function` is expected to be an Embeddings object, support for passing in a function will soon be removed.
```

- 실행 시간 (8.2s)
- `embedding_function` is expected to be an Embeddings object, support for passing in a function will soon be removed.

```
# ========================================
# 5. 메모리 검색 테스트
```

```
# ========================================
print("질문: 면접자 전공은 무엇인가요?", "\n")
print("검색 결과:", memory.load_memory_variables({"prompt": "면접자 전공은 무엇인가요?"})["history"])
print("\n", "==="*50, "\n")
print("질문: 면접자가 프로젝트에서 맡은 역할은 무엇인가요?", "\n")
print("검색 결과:", memory.load_memory_variables({"human": "면접자가 프로젝트에서 맡은 역할은 무엇인가요?"})["history"]
```

- 셀 출력 (0.7s)

```
질문: 면접자 전공은 무엇인가요?

검색 결과: human: 프로젝트에서 어떤 역할을 맡았나요?
ai: 제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을 담당했
```

```
==================================================================================================================================
```

```
질문: 면접자가 프로젝트에서 맡은 역할은 무엇인가요?

검색 결과: human: 프로젝트에서 어떤 역할을 맡았나요?
ai: 제가 맡은 역할은 백엔드 개발자였습니다. 사용자 데이터 처리와 서버 로직 개발을 담당했으며, RESTful API를 구현하여 프론트엔드와의 통신을 담당했
```

```
# ========================================
# 6. (선택) gpt-4o-mini와 결합 예시
# ========================================
# OpenAI API 키가 있어야 동작합니다. 없으면 이 부분은 주석 처리하세요.
# .env 파일에서 API 키 불러오기
load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")

llm = ChatOpenAI(
    model="gpt-4o-mini",
    api_key=api_key)

question = "이 면접자의 강점을 한 문장으로 요약해줘."
context = memory.load_memory_variables({"human": question})["history"]

response = llm.invoke(f"다음 대화 기록을 참고해서 답변해줘:\n{context}\n\n질문: {question}")
print("\nGPT 응답:", response.content)
```

- 셀 출력 (2.4s)

```
GPT 응답: 면접자는 팀 내 의사소통 문제를 해결하기 위해 정기적인 미팅과 의견 공유를 적극적으로 활용한 강점을 가지고 있습니다.
```

- *next: LCEL Chain 에 메모리 추가*
