Retrieval-Augmented Generation (RAG)

단국대학교, 2025

1. RAG란 무엇인가?
2. 벡터 DB(Vector Database)란?
3. 실습 데이터셋 선정 및 준비
4. LangChain 핵심 모듈 심층 분석
    - Embedding, Retriever
5. 단계별 실습 코드
    - 데이터셋 임베딩
    - FAISS 벡터 DB 구축
    - Retriever 및 RAG 파이프라인 구현
6. RAG 적용/미적용 비교 실험
    - 결과 및 차이점 분석

강의 목표:

- GPT4All 모델 기반 RAG 시스템 구현
- RAG 적용 전후 성능 비교 분석 방법

RAG란 무엇인가?

생성형 AI의 한계와 RAG의 필요성

- **생성형 AI(LLM)**는 사전 학습된 데이터만을 바탕으로 답변을 생성합니다.
- 최신 정보, 특정 도메인(사내 정책, 논문, 법률 등)에 대한 질문에는
  부정확하거나 “환각(hallucination)” 답변을 생성할 수 있습니다.
- 예시:
  - “2024년 회사 휴가 정책이 어떻게 바뀌었나요?”
  - LLM은 사내 문서를 알지 못해, 틀리거나 추측성 답변을 할 위험이 큽니다.

RAG(Retrieval-Augmented Generation)의 개념

- **RAG**는 생성형 AI가 외부 지식(문서, 데이터베이스 등)을
  실시간으로 검색하여, 그 결과를 바탕으로 답변을 생성하는 구조입니다.
- 즉, “검색”과 “생성”을 결합하여
  최신 정보와 신뢰성 있는 답변을 제공합니다.

> "RAG는 LLM의 두뇌에 외부 기억 장치를 연결하는 기술"

RAG의 동작 구조

1. **질문 입력**
    - 사용자가 자연어로 질문을 입력합니다.
2. **임베딩(Embedding)**
    - 질문을 고차원 벡터로 변환합니다.
3. **벡터 DB 검색**
    - 임베딩 벡터를 기반으로, 의미적으로 유사한 문서/문단을 검색합니다.
4. **LLM 답변 생성**
    - 검색된 문서를 LLM에 함께 입력하여,
      근거 기반의 답변을 생성합니다.

모델 선택 기준:

- 용량에 따른 최적화
  - 4GB: `Phi-3-mini-4k-instruct.Q4_0.gguf` (2.18GB)
  - 8GB: `Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf` (4.11GB)
  - 16GB: `gpt4all-13b-snoozy-q4_0.gguf` (7.37GB)
- 작업 유형별 추천
  - 대화, 문서요약, 질의응답 등등...
  - 학습한 데이터를 보고 고를 수 있다!

모델 고르기

- 저번시간에 설명한대로 일단 다 돌려보고 고를 수도 있지만...
- 모델이 '잘' 작동하는 분야가 다르다는걸 확인하려면 어떤 데이터에 대해서 얼마나 잘하는지 확인 해볼 수 있습니다.

모델 고르기

- `huggingface.co` 에서 모델을 검색해봅니다. 보통은 gpt4all 에서 사용하는 모델들은 'quantized' 되어있을 뿐이지, 그 원본 모델이 존재합니다.
- `Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf` 모델을 예로 들어봅시다

`Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf`

- GPT4all 이 올린건 아니지만, 같은 이름의 모델을 찾을 수 있습니다.

Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf

- 들어가서 model card 탭을 보면 어떤 모델을 기반으로 gguf로 변환했는지 확인됩니다.
- `NousResearch/Nous-Hermes-2-Mistral-7B-DPO` 눌러서 다시 들어갑니다.

Model Card

- model card는 업로더가 모델에 대한 정보를 적어놓은 곳이기 때문에, 모델의 성능이 어떤지 - 어떤 데이터로 학습을 했는지, 어떤 프롬프트를 쓰면 좋은지와 같은 정보가 나와있습니다.

- 다양한 벤치마크가 있지만, 여기서는 Big Bench 기준오로 봅시다
  - **Big bench**: 다양한 자연어 처리 작업들에서의 성능을 평가하도록 되어있는 벤치마크

Task 이름	테스트하는 능력 (간단 설명)
bigbench_causal_judgement	인과관계 및 도덕적 책임 판단 능력 평가
bigbench_date_understanding	날짜 계산 및 날짜 추론 능력 평가
bigbench_disambiguation_qa	대명사의 정확한 지시 대상 파악 능력 평가
bigbench_geometric_shapes	기하학적 도형의 특징, 관계 등 도형 인식 및 논리 추론 평가
bigbench_logical_deduction_five_objects	5개 객체에 대한 논리적 추론 및 연역 능력 평가
bigbench_logical_deduction_seven_objects	7개 객체에 대한 논리적 추론 및 연역 능력 평가
bigbench_logical_deduction_three_objects	3개 객체에 대한 논리적 추론 및 연역 능력 평가

(계속)

Task 이름	테스트하는 능력 (간단 설명)
bigbench_movie_recommendation	영화 추천 관련 추론 능력 평가
bigbench_navigate	경로 안내, 지도 읽기 등 공간적 추론 및 내비게이션 능력 평가
bigbench_reasoning_about_colored_objects	색상, 개수, 조합 등 여러 색상의 사물에 대한 추론 능력 평가
bigbench_ruin_names	고대 유적지나 장소 이름에 대한 상식 또는 추론 능력 평가
bigbench_salient_translation_error_detection	번역문에서 눈에 띄는 오류 탐지 능력 평가
bigbench_snarks	언어 유희, 수수께끼, 말장난 등 언어적 유연성 및 창의적 사고 평가

(계속)

Task 이름	테스트하는 능력 (간단 설명)
bigbench_sports_understanding	스포츠 규칙, 경기 상황 등 스포츠 관련 상식 및 추론 능력 평가
bigbench_temporal_sequences	시간적 순서, 사건의 전후 관계 등 시간 추론 능력 평가
bigbench_tracking_shuffled_objects_five_objects	5개 객체 셔플 후 위치 추적 및 기억 능력 평가
bigbench_tracking_shuffled_objects_seven_objects	7개 객체 셔플 후 위치 추적 및 기억 능력 평가
bigbench_tracking_shuffled_objects_three_objects	3개 객체 셔플 후 위치 추적 및 기억 능력 평가

내가 만들고 싶은 챗봇이 가져야하는 능력을 기준으로 모델을 고르면 됩니다.

- 예를 들어, 추론을 잘하는 모델을 원한다면, `bigbench_logical_deduction_*_objects` 같은 task를 잘하는 모델을 고르면 됩니다.

벡터 DB(Vector Database)란?

전통 DB와 벡터 DB의 차이

구분	전통 DB(RDB)	벡터 DB
데이터	테이블(행/열)	고차원 벡터(숫자 배열)
검색	정확 일치, SQL 쿼리	유사도 기반(거리 계산)
활용	구조화 데이터 관리	NLP, 추천, 검색 등
확장성	제한적	대규모 분산 지원

- **전통 DB**는 “정확히 일치하는 값” 검색에 최적화되어 있습니다.
- **벡터 DB**는 “비슷한 의미”의 데이터를 벡터 공간에서
  빠르게 검색하는 데 특화되어 있습니다.

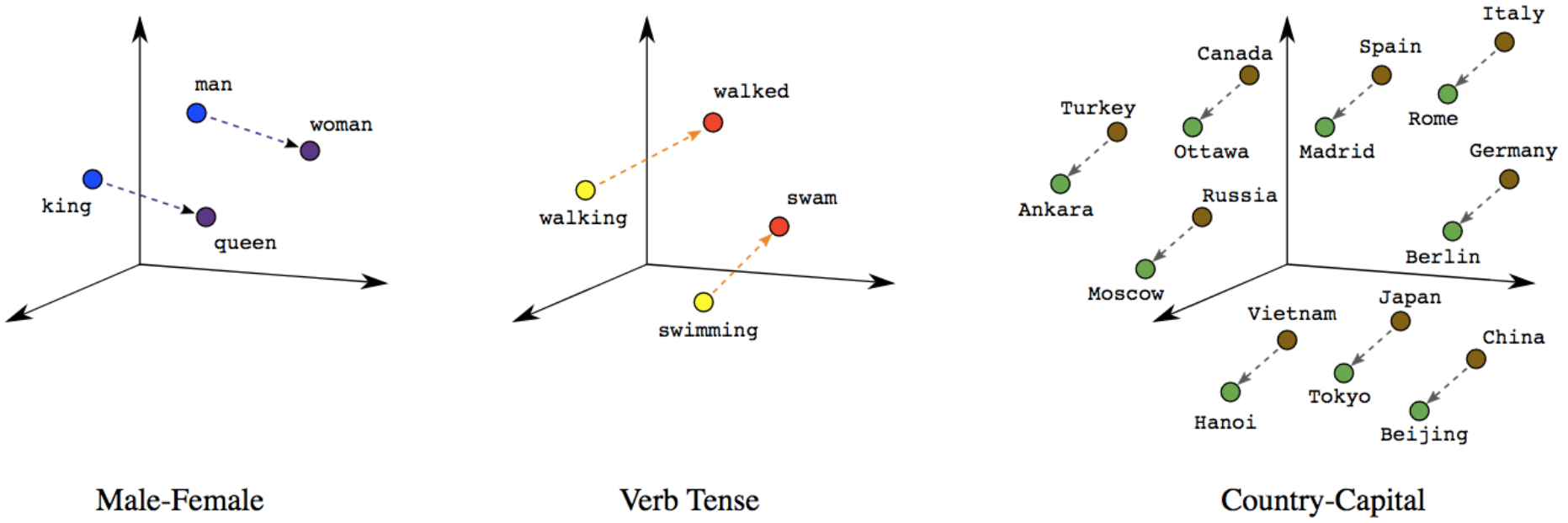
임베딩 (복습)

단어 임베딩

- 컴퓨터가 이해할 수 있는 형태로 단어를 넣어주기 위해서는 결국 숫자로 표현해야한다.
- 각 단어를 어떤 숫자로 표현할건지 생각해보다보면 문제가 생긴다.
  - 1: 사과, 2: 바나나, 3: 오렌지, 4:포도 ... 이런식으로 표현한다면
  - 예를 들어, 사과-바나나가 비슷한 정도가 오렌지-포도가 비슷한 정도랑 같다고 이야기할 수 없다.
- 그래서 벡터의 형태로 표현하도록 만든다.
  - 사과: [0.1, 0.2, 0.3, 0.2, 0.2]
  - 바나나: [0.2, 0.3, 0.4, 0.3, 0.1]
  - 강아지: [0.9, 0.2, 0.2, 0.1, 0.1]
  - 이런식으로 표현하면, 사과와 바나나가 비슷한 정도를 이야기할 수 있다.

임베딩 (복습)

단어 임베딩



Male-Female　　Verb Tense　　Country-Capital

- 비슷한 단어는 비슷한 위치에 존재하도록 만든다.

벡터 DB가 필요한 이유

- 자연어 질의는 “동의어”, “오타”, “문장 구조 변화” 등
  다양한 표현을 포함합니다.
- 단순 키워드 검색은
  - “휴가”라는 단어가 없으면 관련 내용을 찾지 못함
  - 의미적으로 유사한 문장도 놓칠 수 있음
- **벡터 DB**는
  - **문장 전체의 의미**를 벡터로 표현 (단어가 아니다!)
  - “비슷한 의미”의 문장도 효과적으로 검색 (예: "연차"라는 단어도 검색 대상으로...)
  - 대규모 데이터셋(수십만~수억 건)에서도 빠른 유사도 검색 가능

벡터 DB의 종류 및 비교

DB명	주요 특징	확장성/성능	통합성/지원
FAISS	고속 검색, GPU 지원, 대규모 처리	대용량, GPU 가속	Python/Numpy
Chroma	LLM 앱 개발 최적화, 쉬운 통합	중소규모, 빠른 처리	LangChain
Pinecone	완전관리형, 높은 확장성	클라우드, 저지연	Python
Milvus	분산처리, AI/ML 연동, 초대규모	수십억 벡터, 분산	다양한 언어
Qdrant	실시간, 고정밀, Rust 기반	클라우드/온프레미스	다언어 API
Weaviate	그래프+벡터, GraphQL 지원	대규모, 유연성	GraphQL
PGVector	PostgreSQL 확장, SQL 통합	RDB+벡터 통합	SQL

FAISS와 Chroma: Python 환경에서 자주 쓰입니다.

Python 기반의 벡터 검색 및 AI 프로젝트에서 **FAISS**와 **Chroma**가 유용하게 통합될 수 있기 때문에 많이 쓰입니다.

FAISS: 고성능 대규모 벡터 검색 라이브러리

- **Python 연동**
  - C++로 구현되어 있지만 Python 바인딩 제공
  - NumPy, PyTorch, TensorFlow (딥러닝 아키텍쳐) 등과 쉽게 통합
- **주요 특징**
  - 대규모 데이터셋(수억~수십억 벡터)에서 탁월한 성능
  - GPU 가속 지원으로 초고속 검색 가능
  - 다양한 인덱싱 및 검색 알고리즘 제공(IVF, HNSW 등)
  - 클러스터링, 유사도 검색 등 ML 파이프라인에 최적

Chroma: Python 친화적 벡터 데이터베이스

- **Python 중심 설계**
  - 순수 Python 구현, 설치와 사용이 매우 쉬움
  - pip로 간단히 설치, 친숙한 Python API 제공
- **주요 특징**
  - 벡터+메타데이터 저장, 실시간 검색, 내장 영속성
  - FastAPI 등 웹 프레임워크와 통합이 쉬움
  - TensorFlow, PyTorch, Hugging Face 등과 자연스럽게 연동
  - HNSW, IVF 등 다양한 인덱싱 방식 지원
  - 수평 확장(클러스터링), 다양한 스토리지 백엔드 지원

FAISS vs Chroma: Python 환경에서의 비교

FAISS는 다음과 같은 환경에서 자주 쓰입니다.

- 대용량 데이터셋 처리
- GPU 활용 환경
- 세밀한 인덱스/성능 튜닝이 필요한 경우
- ML 연구 및 대규모 검색 시스템 구축

Chroma는 다음과 같은 환경에서 자주 쓰입니다.

- 빠른 개발, 쉬운 통합이 필요한 Python 기반 AI/LLM 프로젝트
- 실시간 데이터 추가 및 검색, 메타데이터 필터링 필요
- 웹 서비스, 챗봇, 추천 시스템 등 애플리케이션 개발

왜 Chroma를 선택하는가?

> 여기서는 Chroma를 사용합니다.

- **오픈소스, 경량, 빠른 속도**:
  인메모리/로컬 저장, 빠른 벡터 검색, 실시간 응답
- **간단한 API**:
  Python 등 다양한 언어 SDK, LangChain 등과의 손쉬운 통합
- **문서 저장/검색/메타데이터 필터링**:
  벡터+원문+메타데이터 통합 관리, 필터링 검색 지원
- **LLM/RAG에 최적화**:
  임베딩, 검색, 문서 관리까지 한 번에 처리 가능

실습 데이터셋 선정 및 준비

실습 목표와 데이터 선정 기준

- 실습 목표가 정해져야 실제로 RAG를 구현하기 위한 DB를 만들고... 하니까요!
- 실제 오픈 데이터셋을 이용해,
  벡터 DB 구축 → 임베딩 → RAG 파이프라인 전체를 만들어 봅시다!

- **데이터 선정 기준:**
  - 공개/자유롭게 사용 가능
  - 자연어 문서(질문-문맥, 문단 등)로 구성
  - 벡터 DB 구축에 적합한 규모(수천~수만 건)
  - Hugging Face, Kaggle 등에서 쉽게 다운로드 가능한게 실습에 좋겠죠.
- **그 외에도....**
  - 여러분이 앞서 배웠던 다른 툴들을 이용해서 PDF를 받아서 (Document Loader) 페이지별로 구분하고 (Text Splitter) Vector DB를 만들고... 하는 방법도 있습니다.

추천 오픈 데이터셋 예시

- **SQuAD (Stanford Question Answering Dataset)**
  - 10만 개 이상의 질문-문맥 쌍
  - 자연어 QA 실습에 표준
- **philschmid/finanical-rag-embedding-dataset**
  - 금융 관련된 도메인 데이터를 갖고 있기 때문에
    RAG로 구현한다면 전문적인 정보를 제공할 수 있어지겠죠!
  - 7,000개 질문-문맥 쌍 (NVIDIA SEC 보고서 기반)
  - Hugging Face에서 바로 다운로드 가능

실습 데이터셋 다운로드 예시

- Hugging Face에서 제공하는 데이터셋을 다운로드합니다 (datasets 라이브러리 사용)
- 데이터셋은 Dataset card라고 하는 곳에 설명이 되어 있습니다.

```
from datasets import load_dataset

# Hugging Face에서 금융 QA 데이터셋 다운로드
dataset = load_dataset("philschmid/finanical-rag-embedding-dataset")
train_data = dataset['train']
print(train_data[0])
```

```
{'question':
    'What area did NVIDIA initially focus on before expanding to
    other computationally intensive fields?',
'context':
    'Since our original focus on PC graphics, we have expanded to
    several other large and important computationally intensive fields.'
}
```

- 'question': 질문
- 'context': 해당 질문에 대한 문맥

이런 식으로 구성되어있기 때문에.. 만약에 사용자가 챗봇에 질문을 했을때 여기 있는것과 유사한 질문을 던지면 답을 찾아올 수 있습니다!

RAG를 위한 LangChain 핵심 모듈

Retriever, Embedding

Embedding 이란?

- **임베딩(Embedding)** 은 텍스트(문장, 문단, 질의 등)를 고차원 벡터(숫자 배열)로 변환하는 과정입니다.
- 임베딩 벡터는 의미적으로 유사한 문장끼리 벡터 공간상에서 가깝게 위치합니다.
- 예를 들어,
  - “GPU 시장”과 “그래픽 카드 제조사”는 임베딩 벡터가 비슷하게 나옵니다.
  - “회계 감사”와는 멀리 떨어집니다.

왜 Embedding이 중요한가?

- 단순 키워드 검색은 “동의어”, “문장 구조 변화”, “오타” 등에 취약합니다.
- 임베딩을 사용하면
  - “그래픽카드를 만드는 회사는?”
  - “GPU 제조사는 어디인가?”
    이 두 문장이 벡터 공간에서 가까워져, 의미 기반 검색이 가능합니다.
- **RAG에서 임베딩 품질이 검색 품질을 좌우**합니다.

LangChain에서 Embedding 사용 예시

```
from langchain_community.embeddings import GPT4AllEmbeddings

embeddings = GPT4AllEmbeddings(model_name = "all-MiniLM-L6-v2.gguf2.f16.gguf")
text = "NVIDIA designs GPUs for gaming and AI."
vector = embeddings.embed_query(text)
print(f"임베딩 벡터 차원: {len(vector)}")
print(f"임베딩 벡터 일부: {vector[:5]}")
```

- 이 코드는 텍스트를 GPT4All 임베딩 모델로 벡터화합니다.
- `GPT4AllEmbeddings` 는 LangChain에서 제공하는 임베딩 모델로, GPT4All 모델을 기반으로 임베딩을 수행하는
- 여기서 `model_name` 은 사용할 모델의 이름을 지정합니다. 그냥 아무 GPT4All 모델이나 넣을 수 있는게 아니라, 임베딩을 목적으로 학습된 '임베딩 모델'을 넣어야 합니다.

임베딩 모델과 언어 생성 모델이 달라도 되나요?

- 예를 들어, 임베딩 모델은 `gpt4all-13b-snoozy-q4_0.gguf` 모델을 사용하고, 언어생섬모델은 `Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf` 이나 `Phi-3-mini-4k-instruct.Q4_0.gguf` 같은 모델을 사용할 수 있습니다.
- 그 역할을 해주는게 Retriever입니다.
  임베딩 모델로 문서와 쿼리를 벡터화해서 유사 문서를 찾고 (Retriever),
  그 문서들을 언어 생성모델에 넣어서 답변을 생성합니다.

Retriever란?

- **Retriever**는 사용자의 질문(임베딩 벡터)과 가장 유사한 문서(임베딩 벡터)를 벡터 DB에서 찾아 주는 역할을 합니다.
- 단순 키워드 매칭이 아니라, **의미적 유사성**(코사인 유사도 등)을 기반으로 검색합니다.

Retriever가 중요한 이유

- 예를 들어,
  사용자가 “GPU를 만드는 회사는 어디인가요?”라고 물었을 때
  - “GPU”라는 단어만 찾는 것이 아니라
  - “그래픽카드”, “비디오카드” 등 의미적으로 관련된 문서도 함께 찾아야 합니다.
  - 예: "그래픽카드를 만드는 회사는 어디인가요?"
- Retriever가 없다면, LLM은 사내 정책이나 최신 정보를 알지 못하거나,
  환각(hallucination) 답변을 할 위험이 큽니다.

LangChain에서 Retriever 사용 예시

```
from langchain.docstore.document import Document

docs = [
    Document(
        page_content=item['context'],
        metadata={'question': item['question']}
    )
    for item in train_data
    if item.get('context') not in [None, ''] and item.get('question') not in [None, '']
]
```

```python
from langchain_community.vectorstores import Chroma

# 문서 리스트를 Chroma에 저장(임베딩 벡터 포함)
db = Chroma.from_documents(docs, embeddings)

# Retriever 생성
retriever = db.as_retriever()

# 질문에 대해 관련 문서 검색
query = "When did NVIDIA release their first GPU?"
results = retriever.get_relevant_documents(query)
for doc in results:
    print("검색된 문서:", doc.page_content[:200])
```

- 사용자의 질문을 임베딩 벡터로 변환 후,
  벡터 DB에서 의미적으로 가장 가까운 문서를 찾아 출력합니다.
- `Document` 객체는 LangChain에서 제공하는 문서 객체로, Document Loader 등을 통해서 반환되는 객체도 이 클래스로 만들어져있습니다.
- `results = retriever.get_relevant_documents(query)` 를 통해서 리트리버는 query를 임베딩 벡터로 변환하고, 만들어진 벡터DB에서 가장 유사한 문서들을 찾아서 반환합니다

- 그 이후 여러 모듈(Embedding, Retriever, LLM 등)을 chain 으로 연결해서
- 하나의 파이프라인으로 연결하여,
  질문 → 검색 → 답변 생성을 자동화합니다!

RAG 파이프라인 구현

```
from langchain_community.llms import GPT4All
from langchain.chains import RetrievalQA

llm = GPT4All(model="Phi-3-mini-4k-instruct.Q4_0.gguf")

qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True
)

question = "What is the main business of NVIDIA?"
result = qa_chain({"query": question})

print("답변:", result['result'])
print("근거 문서:", [doc.page_content[:200] for doc in result['source_documents']])
```

RetrievalQA

- `RetrievalQA`는 LangChain에서 제공하는 질의응답 체인으로,
- LLM과 Retriever를 연결하여 질문에 대한 답변을 생성하도록 자동으로 chain을 구성합니다.
- `chain_type`은 여러 종류가 있지만, `stuff`는 관련된 문서들을 모두 prompt에 채워 넣어(stuff) 전달한다는 뜻.
- 물론 이전에 chain을 만들던 대로 만들수도 있습니다.

그러려면 모델의 프롬프트를 확인해봅시다.

https://huggingface.co/microsoft/Phi-3-mini-4k-instruct-gguf

```
from langchain.prompts import PromptTemplate


prompt_template = """<|system|>
You are a helpful assistant.<|end|>
<|user|>
Context:
{context}

Question:
{question}<|end|>
<|assistant|>
"""

prompt = PromptTemplate(
    template=prompt_template,
    input_variables=["context", "question"]
)

chain = prompt | llm

question = "What is the main business of NVIDIA?"
context = "\n\n".join([doc.page_content for doc in retriever.get_relevant_documents(question)])

result = chain.invoke({"context": context, "question": question})
print("RAG 답변:", result)
print("근거 문서:", context[:200])
```

RAG 미적용: LLM 단독 질의응답

```
# 컨텍스트 없이 동일한 chat format 프롬프트 사용
single_prompt_template = """<|system|>
You are a helpful assistant.<|end|>
<|user|>
Question:
{question}<|end|>
<|assistant|>
"""

single_prompt = PromptTemplate(
    template=single_prompt_template,
    input_variables=["question"]
)

chain_no_rag = single_prompt | llm

result_no_rag = chain_no_rag.invoke({"question": question})
print("LLM 단독 답변:", result_no_rag)
```

RAG 적용한 경우

- 질문 → 임베딩 → Chroma에서 의미적으로 유사한 문서 검색 →
  LLM이 근거 문서와 함께 답변 생성.
  답변의 신뢰성과 정확성이 크게 향상됩니다.

RAG 미적용한 경우

- LLM만 사용할 경우,
  사전 학습된 정보에만 의존하여 답변을 생성합니다.
  최신 정보, 특정 도메인(예: SEC 보고서 기반 실제 데이터)에는
  할루시네이션(환각)이나 부정확한 답변이 나올 수 있습니다.

RAG 적용/미적용 결과 비교

항목	LLM 단독 (RAG 미적용)	RAG 적용 (Chroma + Retriever + LLM)
정확도	낮음 (환각, 추측성 답변)	높음 (문서 근거 기반)
신뢰성	낮음 (출처 불명, 불확실)	높음 (출처 명확, 문맥 제공)
최신성	학습 시점 이후 정보 없음	최신 문서 반영 가능
근거 제시	불가	가능 (근거 문서 함께 제공)
유연성	정적(모델 재학습 필요)	동적(문서만 교체해도 최신화)

실제 답변 예시 비교

질문:
What is the main business of NVIDIA?

- **LLM 단독 답변 예시:**

> "NVIDIA is a technology company known for its graphics processing units (GPUs), primarily used in gaming and professional markets..."

- **RAG 적용 답변 예시:**

> "NVIDIA's main business is designing and selling GPUs for gaming, data centers, and AI workloads. According to the 2022 SEC report:
> 'NVIDIA's primary business is the design and sale of graphics processing units (GPUs) for gaming, professional visualization, data center, and automotive markets.'"

- **근거 문서:**

결론

- **RAG는 LLM의 한계를 극복하는 가장 효과적인 방법**입니다.
- 실시간, 신뢰성, 최신성, 근거 제시 등
  실제 업무·서비스에서 요구되는 조건을 충족시킬 수 있습니다.
- 벡터 DB, Retriever, Embedding 등 각 구성요소의 품질이
  전체 시스템 성능에 직접적인 영향을 미치므로
  설계·구현 단계에서 꼼꼼한 검증이 필요합니다.

참고 자료

- LangChain 공식 문서
- Chroma 공식 문서
- GPT4All 공식 깃허브
- Hugging Face Datasets
- RAG 논문 (Lewis et al., 2020)

감사합니다!