

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

2. LangChain의 RAG 파헤치기

RAG 프로세스

문서 로드 및 Vector DB 저장

| | | |
|---|--------------------|--|
| 1 | 문서 로드 (Load) | 문서(pdf, word), RAW DATA, 웹페이지, Notion 등의 데이터를 읽기 |
| 2 | 분할 (Split) | 불러온 문서를 chunk 단위로 분할 |
| 3 | 임베딩 (Embedding) | 문서를 벡터 표현으로 변환 |
| 4 | 벡터DB (VectorStore) | 변환된 벡터를 DB에 저장 |

RAG 프로세스

문서 검색 및 결과 도출

| | | |
|---|----------------|---|
| 5 | 검색 (Retrieval) | 유사도 검색(similarity, mmr), Multi-Query, Multi-Retriever |
| 6 | 프롬프트 (Prompt) | 검색된 결과를 바탕으로 원하는 결과를 도출하기 위한 프롬프트 |
| 7 | 모델 (LLM) | 모델 선택 (GPT-3.5, GPT-4, etc) |
| 8 | 결과 (Output) | 텍스트, JSON, 마크다운 |

1) 질문 처리

- 질문 처리**: 사용자의 질문을 받아 이를 처리 → 관련 데이터를 찾는 작업
- 필요한 구성 요소:
 - 데이터 소스 연결**:
 - 질문에 대한 답변을 찾기 위해 다양한 텍스트 데이터 소스에 연결해야 함
 - LangChain** = 다양한 데이터 소스와의 연결을 간편하게 설정할 수 있도록 도움
 - 데이터 인덱싱 및 검색**:
 - 데이터 소스에서 관련 정보를 효율적으로 찾기 위해, 데이터는 인덱싱되어야 함
 - LangChain** = 인덱싱 과정을 자동화하고, 사용자의 질문과 관련된 데이터를 검색하는 데 필요한 도구를 제공



2) 답변 생성

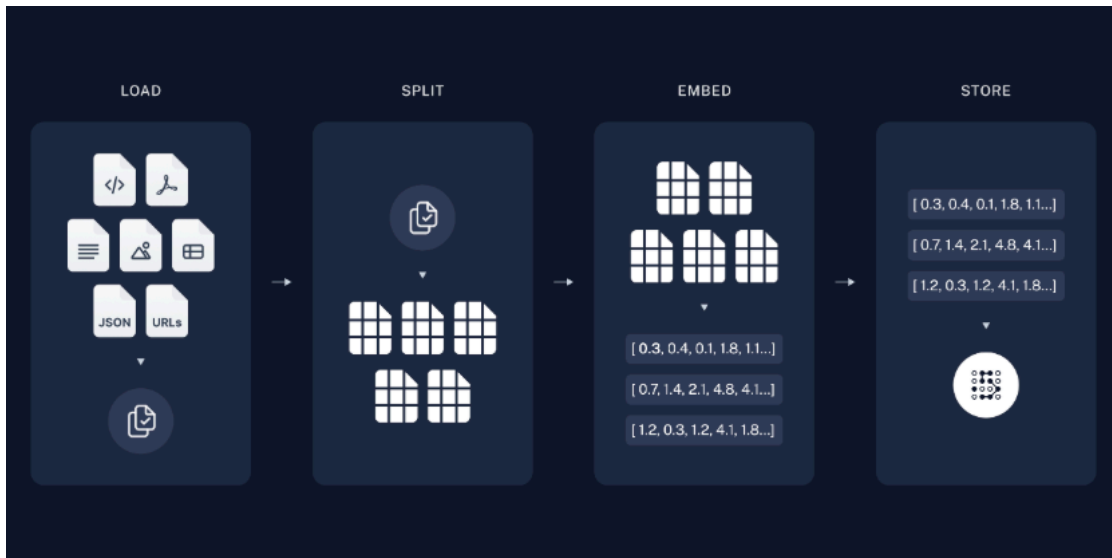
- 관련 데이터를 찾은 후 → 사용자의 질문에 **답변 생성**
 - **중요** 구성 요소:
 - **답변 생성 모델**:
 - LangChain = 고급 자연어 처리 (NLP) 모델 사용 → 검색된 데이터로부터 **답변**을 **생성**할 수 있는 기능 제공
 - 사용자의 질문과 검색된 데이터를 입력 → **적절한 답변**을 **생성**
-

3) 아키텍처

- **Q&A소개** → RAG app 구현해보기
 - 주요 구성 요소
 - **인덱싱**:
 - 소스에서 데이터를 수집하고 인덱싱하는 파이프라인
 - 이 작업은 **보통 오프라인에서 발생**
 - **검색 및 생성**:
 - 실제 **RAG 체인**
 - 사용자 쿼리를 실행 시간에 받아 **인덱스**에서 **관련 데이터**를 **검색** → 그 데이터를 **모델**에 **전달**
-

4) 인덱싱

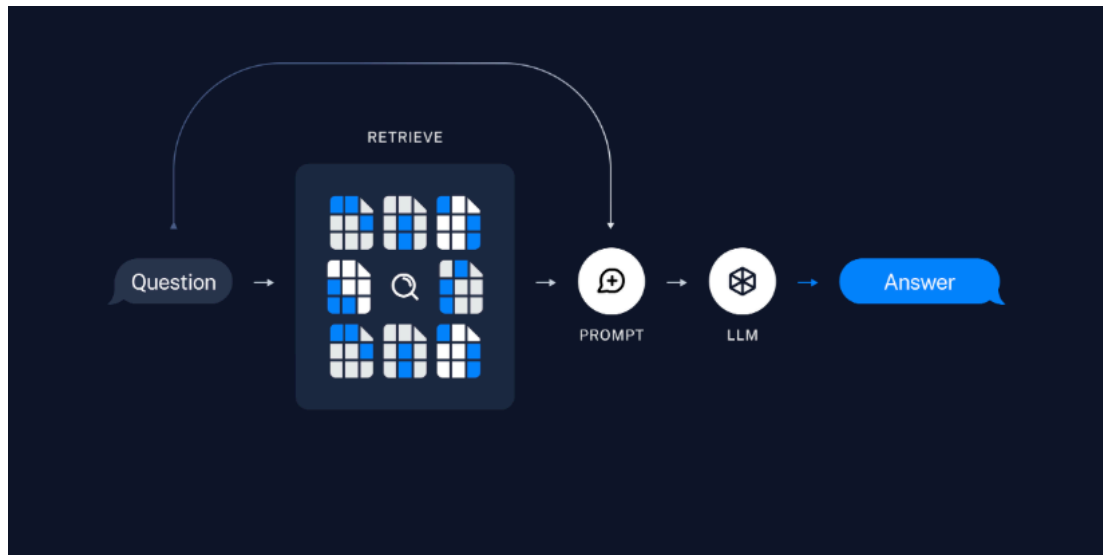
- **process_1**



- a. **로드**:
 - 먼저 데이터를 로드해야 함
 - **DocumentLoaders** 사용
- b. **분할**:
 - **Text splitters** = 큰 Documents를 더 작은 청크로 나눔
 - 데이터를 인덱싱하고 모델에 전달하는 데 **유용** = 큰 청크는 검색하기 어렵고 모델의 유한한 컨텍스트 창에 맞지 않음
- c. **저장**:
 - 나중에 검색할 수 있도록 분할을 저장하고 인덱싱할 장소가 필요
 - **VectorStore**, **Embeddings 모델** 사용 → 수행

5) 검색 및 생성

- process_2



- - d. 검색 :
 - 사용자 입력 → **Retriever** 를 사용하여 저장소에서 관련 분할을 검색함
 - e. 생성 :
 - **ChatModel** / **LLM** = 질문 + 검색된 데이터를 포함한 **프롬프트** 를 사용 → **답변 생성**

- **실습 활용 문서**
 - **소프트웨어정책연구소 (SPRi) - 2023년 12월호**
 - 저자: 유재흥(AI정책연구실 책임연구원), 이지수(AI정책연구실 위촉연구원)
 - 링크: <https://spri.kr/posts/view/23669>
 - 파일명: [SPRI AI Brief 2023년12월호 F.pdf](#)

- **환경설정**

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv
```

```
# API 키 정보 로드
load_dotenv()                                     # True
```

```
from langsmith import Client
from langsmith import traceable

import os

# 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_API_KEY') else "설정되지 않음" # API 키 값은 직접 출력하지 않음

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langchain_project:
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='{langchain_tracing_v2}')

```

- 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.
```

3) RAG 기본 파이프라인 (1~8단계)

```
import bs4
from langchain import hub
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma, FAISS
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import PromptTemplate
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_huggingface import HuggingFaceEmbeddings
```

- USER_AGENT environment variable not set, consider setting it to identify your requests. (3.6s)

```
# 단계 1: 문서 로드(Load Documents)
# 뉴스기사 내용을 로드, 청크 나누기, 인덱싱
url = "https://n.news.naver.com/article/437/0000378416"

loader = WebBaseLoader(
    web_paths=(url,),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            "div",
            attrs={"class": ["newsct_article _article_body", "media_end_head_title"]},
        )
    ),
)

docs = loader.load()                                # 0.4s
```

```
print(type(docs))                                # <class 'list'>
```

```
# 단계 2: 문서 분할(Split Documents)

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50)
splits = text_splitter.split_documents(docs)
print(f"분할된 청크의수: {len(splits)}")
```

- 분할된 청크의수: 3

```
# 단계 3: 임베딩(Embedding) 생성
from langchain_huggingface import HuggingFaceEmbeddings
import warnings

# 경고 무시
warnings.filterwarnings("ignore")

# HuggingFace Embeddings 사용
embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
)

print("✅ hugging-face 임베딩 모델 로딩 완료!")
```

- ✅ hugging-face 임베딩 모델 로딩 완료! (6.4s)

```
# 단계 4: 벡터스토어 생성하기
vectorstore = FAISS.from_documents(
    documents=splits,
    embedding=embeddings
)

print("🎉 Huggingface 모델을 사용하여 로컬에서 벡터스토어(FAISS)가 성공적으로 생성되었습니다!")
```

- 🚀 Huggingface model을 사용하여 로컬에서 벡터스토어(FAISS)가 성공적으로 생성되었습니다! (0.3s)

```
# 단계 5: 검색(Search)
# 뉴스에 포함되어 있는 정보를 검색하고 생성하기
retriever = vectorstore.as_retriever()
```

```
# 검색기에 쿼리를 날려 검색된 chunk 결과 확인하기
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"      # 병렬처리 비활성화
```

```
# 단계 6: 프롬프트 생성(Create Prompt)
# 프롬프트 생성하기
prompt = hub.pull("rlm/rag-prompt")
```

```
print(prompt)
```

- print(prompt) 확인

```
input_variables=['context', 'question'] input_types={} partial_variables={} metadata={'lc_hub_owner': 'rlm', 'lc_
```

```
# 단계 7: 언어모델(LLM) 생성
# 모델(LLM) 생성하기
from langchain_google_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv
import os

load_dotenv()

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-lite",
    temperature=0,                      # temperature = 0으로 설정
    max_output_tokens=4096,
)
```

- LLM 생성하기

```
E0000 00:00:1759663618.303843 752051 alts_credentials.cc:93] ALTS creds ignored. Not running on GCP and untrusted
```

```
def format_docs(docs):
    # 검색한 문서 결과를 하나의 문단으로 합쳐서 출력
    return "\n\n".join(doc.page_content for doc in docs)
```

```
# 단계 8: 체인(Chain) 생성
rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | gemini_lc
    | StrOutputParser()
)
```

- 생성된 체인에 쿼리 (질문)을 입력 → 실행하기

```
# 단계 9: 체인 실행(Run Chain)
# 문서에 대한 질의를 입력하고, 답변을 출력하기
question = "부영그룹의 출산 장려 정책에 대해 설명해주세요"
response = rag_chain.invoke(question)

# 결과 출력
print(f"URL: {url}")
print(f"문서의 수: {len(docs)}")
print("==" * 20)
print(f"[HUMAN]\n{question}\n")
print(f"[AI]\n{response}")
```

- 체인 실행 결과 (1.1s)

URL: <https://n.news.naver.com/article/437/0000378416>

문서의 수: 1

[HUMAN]

부영그룹의 출산 장려 정책에 대해 설명해주세요

[AI]

부영그룹은 2021년 이후 태어난 직원 자녀에게 1억원을 지원하는 파격적인 출산 장려 정책을 발표했습니다. 또한, 셋째 자녀를 출산하는 직원에게는 국민주택

3. 단계 1: 문서 로드 (Load Documents)

- 참고: [공식문서 링크 - Document loaders](#)

1) 웹페이지

- WebBaseLoader** = 저장된 웹페이지에서 필요한 부분만을 파싱하기 위해 **bs4.SoupStrainer** 사용
 - bs4.SoupStrainer** = 편리하게 웹에서 원하는 요소를 가져올 수 있도록 해줌

```
bs4.SoupStrainer(
    "div",
    attrs={"class": ["newsct_article _article_body", "media_end_head_title"]}, # 클래스 명을 입력
)

bs4.SoupStrainer(
    "article",
    attrs={"id": ["dic_area"]}, # 클래스 명을 입력
)
```

```
# test - BBC 기사

loader = WebBaseLoader(
    web_paths=("https://www.bbc.com/news/business-68092814",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            "main",
            attrs={"id": ["main-content"]},
        ),
    ),
)

docs = loader.load()
print(f"문서의 수: {len(docs)}")
docs[0].page_content[:500]
```

- 문서의 수: 1 (1.8s)

'Could AI \'trading bots\' transform the world of investing?1 February 2024ShareSaveJonty BloomBusiness reportersS

2) PDF

```
from langchain.document_loaders import PyPDFLoader

# PDF 파일 로드. 파일의 경로 입력
loader = PyPDFLoader("../12_RAG/data/SPRI_AI_Brief_2023년12월호_F.pdf")

# 페이지 별 문서 로드
docs = loader.load()
print(f"문서의 수: {len(docs)}")

# 10번째 페이지의 내용 출력
print(f"\n[페이지내용]\n{docs[10].page_content[:500]}")
print(f"\n[metadata]\n{docs[10].metadata}\n")
```

- 문서의 수: 23 (2.3s)

[페이지내용]

SPRi AI Brief | 2023-12월호

8

코히어, 데이터 투명성 확보를 위한 데이터 출처 탐색기 공개n코히어와 12개 기관이 광범위한 데이터셋에 대한 감사를 통해 원본 데이터 출처, 재라이선스 KEY Contents

데이터 출처 탐색기, 광범위한 데이터셋 정보 제공을 통해 데이터 투명성 향상nAI 기업 코히어(Cohere)가 매사추세츠 공과대(MIT), 하버드대 로스쿨, 카

[metadata]

{'producer': 'Hancom PDF 1.3.0.542', 'creator': 'Hwp 2018 10.0.0.13462', 'creationdate': '2023-12-08T13:28:38+09:00'}

3) CSV

- CSV 는 페이지 번호 대신 행번호 로 데이터 조회하기

```
from langchain_community.document_loaders.csv_loader import CSVLoader
```

```
# CSV 파일 로드
```

```
loader = CSVLoader(file_path="../12_RAG/data/titanic.csv")
```

```
docs = loader.load()
```

```
print(f"문서의 수: {len(docs)}")
```

```
# 10번째 페이지의 내용 출력
```

```
print(f"\n[페이지내용]\n{docs[10].page_content[:500]}")
```

```
print(f"\n[metadata]\n{docs[10].metadata}\n")
```

- 문서의 수: 891 (0.0s)

[페이지내용]

PassengerId: 11

Survived: 1

Pclass: 3

Name: Sandstrom, Miss. Marguerite Rut

Sex: female

Age: 4

SibSp: 1

Parch: 1

Ticket: PP 9549

Fare: 16.7

Cabin: G6

Embarked: S

[metadata]

{'source': '../12_RAG/data/titanic.csv', 'row': 10}

4) TXT

```
from langchain_community.document_loaders import TextLoader
```

```
loader = TextLoader("../12_RAG/data/appendix-keywords.txt")
```

```
docs = loader.load()
```

```
print(f"문서의 수: {len(docs)}")
```

```
# 10번째 페이지의 내용 출력
```

```
print(f"\n[페이지내용]\n{docs[0].page_content[:500]}")
```

```
print(f"\n[metadata]\n{docs[0].metadata}\n")
```

- 문서의 수: 1 (0.0s)

[페이지내용]
Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환하는 검색 방식입니다.
예시: 사용자가 "태양계 행성"이라고 검색하면, "목성", "화성" 등과 같이 관련된 행성에 대한 정보를 반환합니다.
연관키워드: 자연어 처리, 검색 알고리즘, 데이터 마이닝

Embedding

정의: 임베딩은 단어나 문장 같은 텍스트 데이터를 저차원의 연속적인 벡터로 변환하는 과정입니다. 이를 통해 컴퓨터가 텍스트를 이해하고 처리할 수 있게 합니다.
예시: "사과"라는 단어를 [0.65, -0.23, 0.17]과 같은 벡터로 표현합니다.
연관키워드: 자연어 처리, 벡터화, 딥러닝

Token

정의: 토큰은 텍스트를 더 작은 단위로 분할하는 것을 의미합니다. 이는 일반적으로 단어, 문장, 또는 구절일 수 있습니다.
예시: 문장 "나는 학교에 간다"를 "나는", "학교에", "간다"로 분할합니다.
연관키워드: 토큰화, 자연어

[metadata]

```
{'source': '../12_RAG/data/appendix-keywords.txt'}
```

5) 폴더 내의 모든 파일 로드

- 폴더 내의 모든 .txt 파일을 로드하는 예제

```
from langchain_community.document_loaders import DirectoryLoader

loader = DirectoryLoader("../12_RAG/data", glob="*.txt", show_progress=True)
docs = loader.load()

print(f"문서의 수: {len(docs)}")

# 10번째 페이지의 내용 출력
print(f"\n\n[페이지내용]\n\n{docs[0].page_content[:500]}")
print(f"\n\n[metadata]\n\n{docs[0].metadata}\n\n")
```

- 상대 경로 수정 → 해당 경로에서 모든 .txt 파일 로드하기 (3.4s)

```
0%|          | 0/6 [00:00<?, ?it/s]libmagic is unavailable but assists in filetype detection. Please consider ins
17%|█         | 1/6 [00:03<00:16, 3.26s/it]libmagic is unavailable but assists in filetype detection. Please con
libmagic is unavailable but assists in filetype detection. Please consider installing libmagic for better results
libmagic is unavailable but assists in filetype detection. Please consider installing libmagic for better results
libmagic is unavailable but assists in filetype detection. Please consider installing libmagic for better results
83%|██████████| 5/6 [00:03<00:00, 1.96it/s]libmagic is unavailable but assists in filetype detection. Please con
100%|██████████| 6/6 [00:03<00:00, 1.76it/s]
```

문서의 수: 6

[페이지내용]
Semantic Search

정의: 의미론적 검색은 사용자의 질의를 단순한 키워드 매칭을 넘어서 그 의미를 파악하여 관련된 결과를 반환하는 검색 방식입니다. 예시: 사용자가 "태양계

Embedding

정의: 임베딩은 단어나 문장 같은 텍스트 데이터를 저차원의 연속적인 벡터로 변환하는 과정입니다. 이를 통해 컴퓨터가 텍스트를 이해하고 처리할 수 있게 합니다

Token

정의: 토큰은 텍스트를 더 작은 단위로 분할하는 것을 의미합니다. 이는 일반적으로 단어, 문장, 또는 구절일 수 있습니다. 예시: 문장 "나는 학교에 간다"를

[metadata]

```
{'source': '../12_RAG/data/appendix-keywords-CP949.txt'}
```


- 폴더 내 모든 `.pdf` 파일 로드하는 예시

```
from langchain_community.document_loaders import DirectoryLoader

loader = DirectoryLoader("../12_RAG/data", glob="*.pdf")
docs = loader.load()

print(f"문서의 수: {len(docs)}\n")
print("[메타데이터]\n")
print(docs[0].metadata)
print("\n===== [앞부분] 미리보기 =====\n")
print(docs[0].page_content[2500:3000])
```

- Warning: No languages specified, defaulting to English.
- 문서의 수: 1 (`29.1s`)

[메타데이터]

`{'source': '../12_RAG/data/SPRI_AI_Brief_2023년12월호_F.pdf'}`

===== [앞부분] 미리보기 =====

하는 기업에게 안전 테스트 결과와 시스템에 관한

주요 정보를 미국 정부와 공유할 것을 요구하고, AI 시스템의 안전성과 신뢰성 확인을 위한 표준 및

AI 생성 콘텐츠 표시를 위한 표준과 모범사례 확립을 추진

△1026 플롭스(FLOPS, Floating Point Operation Per Second)를 초과하는 컴퓨팅 성능 또는 생물학적 서열 데이터를 주로 사용하고 1023플롭스

n (형평성과 시민권 향상) 법률, 주택, 보건 분야에서 AI의 무책임한 사용으로 인한 차별과 편견 및 기타

문제를 방지하는 조치를 확대

형사사법 시스템에서 AI 사용 모범사례를 개발하고, 주택 임대 시 AI 알고리즘 차별을 막기 위한 명확한

지침을 제공하며, 보건복지

6) Python

```
from langchain_community.document_loaders import PythonLoader

loader = DirectoryLoader("../12_RAG/data", glob="*.py", loader_cls=PythonLoader)
docs = loader.load()

print(f"문서의 수: {len(docs)}\n")
print("[메타데이터]\n")
print(docs[0].metadata)
print("\n===== [앞부분] 미리보기 =====\n")
print(docs[0].page_content[:500])
```

- 문서의 수: 1 (`0.0s`)

[메타데이터]

`{'source': '../12_RAG/data/audio_utils.py'}`

===== [앞부분] 미리보기 =====

```
import re
import os
from pytube import YouTube
from moviepy.editor import AudioFileClip, VideoFileClip
from pydub import AudioSegment
from pydub.silence import detect_nonsilent
```

```
def extract_abr(abr):
    youtube_audio_pattern = re.compile(r"\d+")
    kbps = youtube_audio_pattern.search(abr)
    if kbps:
        kbps = kbps.group()
        return int(kbps)
    else:
        return 0

def get_audio_filepath(filename):
    # audio 폴더가 없으면 생성
    if not os.path.isdir("audio"):
        os.mkdir("au
```

4. 단계 2: 문서 분할 (Split Documents)

```
# 뉴스기사의 내용 로드, 청크 나누기, 인덱싱

loader = WebBaseLoader(
    web_paths=("https://www.bbc.com/news/business-68092814",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            "main",
            attrs={"id": ["main-content"]},
        ),
    ),
)

docs = loader.load()
print(f"문서의 수: {len(docs)}")
docs[0].page_content[:500]
```

- 문서의 수: 1 (0.4s)

'Could AI \'trading bots\' transform the world of investing?1 February 2024ShareSaveJonty BloomBusiness reportersS

1) CharacterTextSplitter

- 가장 간단한 방법 = 문자 기준으로 분할
 - 기본값: "\n\n"
 - 청크 길이 = 문자의 수로 측정
 - 텍스트가 어떻게 분할되는지: 단일 문자 단위
 - 청크 크기가 어떻게 측정되는지: len of characters
 - [시각화 예제](#)
- CharacterTextSplitter 클래스 = 텍스트를 특정 크기의 청크로 분할하는 기능 제공
 - separator 매개변수
 - 청크를 구분하는 데 사용되는 문자열을 지정
 - 두 개의 개행 문자("\n\n")를 사용
 - chunk_size = 각 청크의 최대 길이 결정
 - chunk_overlap = 인접한 청크 간에 겹치는 문자의 수 지정
 - length_function
 - 청크의 길이를 계산하는 데 사용되는 함수
 - 기본적으로 문자열의 길이를 반환하는 len 함수 사용
 - is_separator_regex = separator가 정규 표현식으로 해석될지 여부를 결정하는 불리언 값

```
from langchain.text_splitter import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    separator="\n\n",          # 청크 구분 문자열
    chunk_size=100,           # 청크 최대 길이 (청크 사이즈)
    chunk_overlap=10,         # 청크 겹치기 허용
    length_function=len,      # 청크 길이 계산 함수
    is_separator_regex=False,  # 정규 표현식으로 해석 X
)
```

- **text_splitter** 객체의 **create_documents** 메소드 사용 → 주어진 텍스트 (**state_of_the_union**)를 여러 문서로 분할 → 그 결과를 **texts** 변수에 저장 → 이후 **texts**의 첫 번째 문서를 출력하기

```
# chain of density 논문의 일부 내용을 불러오기

with open("../12_RAG/data/chain-of-density.txt", "r") as f:
    text = f.read()[:500]

text_splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=10,
    separator="\n\n"          # 기본 청크 구분 단위 (문단 단위)
)

text_splitter.split_text(text)
```

- 기본 청크 구분 단위 = **문단 단위**

```
['Selecting the “right” amount of information to include in a summary is a difficult task. \nA good summary should
```

```
text_splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=10,
    separator="\n"          # 청크 구분 문자열 변경 (줄 단위)
)

text_splitter.split_text(text)
```

- 청크 구분 문자열 = **줄 단위**

```
['Selecting the “right” amount of information to include in a summary is a difficult task.',
'A good summary should be detailed and entity-centric without being overly dense and hard to follow. To better un
```

```
text_splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=10,
    separator=" "          # 청크 구분 문자열 변경 (공백 단위)
)

text_splitter.split_text(text)
```

- 청크 구분 문자열 = **공백 단위**

```
['Selecting the “right” amount of information to include in a summary is a difficult task. \nA good',
'A good summary should be detailed and entity-centric without being overly dense and hard to follow.',
'to follow. To better understand this tradeoff, we solicit increasingly dense GPT-4 summaries with',
'with what we refer to as a “Chain of Density” (CoD) prompt. Specifically, GPT-4 generates an initial',
'an initial entity-sparse summary before iteratively incorporating missing salient entities without',
'without increasing the length. Summaries genera']
```

```
text_splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=0,          # 청크 중복 허용 X
    separator=" "          # 청크 구분 문자열 변경 (공백 단위)
)

text_splitter.split_text(text)
```

- 청크 중복 허용 X

```
['Selecting the “right” amount of information to include in a summary is a difficult task. \nA good',
'summary should be detailed and entity-centric without being overly dense and hard to follow. To',
'better understand this tradeoff, we solicit increasingly dense GPT-4 summaries with what we refer to',
'as a “Chain of Density” (CoD) prompt. Specifically, GPT-4 generates an initial entity-sparse summary',
'before iteratively incorporating missing salient entities without increasing the length. Summaries',
'genera']
```

```
text_splitter = CharacterTextSplitter(
    chunk_size=1000,          # 청크 단위 = 크게 잡음
    chunk_overlap=100,        # 청크 중복 크게 허용
    separator=" "             # 청크 구분 문자열 변경 (공백 단위)
)

# text 파일을 청크로 나누기
text_splitter.split_text(text)

# document를 청크로 나누기
split_docs = text_splitter.split_documents(docs)
len(split_docs)           # 8
print(f"분할된 문서의 수: {len(split_docs)}")
```

- 분할된 문서의 수: 8

```
print(type(split_docs))      # <class 'list'>
```

```
split_docs[0]
```

- 새롭게 분할한 문서 내용

```
Document(metadata={'source': 'https://www.bbc.com/news/business-68092814'}, page_content='Could AI \'trading bots
```

```
# 뉴스기사의 내용 로드, 청크 나누기, 인덱싱

loader = WebBaseLoader(
    web_paths=("https://www.bbc.com/news/business-68092814",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            "main",
            attrs={"id": ["main-content"]},
        ),
    ),
)

# splitter 정의하기
text_splitter = CharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100,
    separator=" "
)

# 문서를 로드시 바로 분할까지 수행하기
split_docs = loader.load_and_split(text_splitter=text_splitter)
print(f"문서의 수: {len(docs)}")
docs[0].page_content[:500]
```

- 문서의 수: 1 (2.3s)

```
'Could AI \'trading bots\' transform the world of investing?1 February 2024ShareSaveJonty BloomBusiness reportersS
```

2) RecursiveTextSplitter

- 일반 텍스트에 권장되는 텍스트 분할기
 - 텍스트가 어떻게 분할 규칙: **list of separators**
 - 청크 크기가 어떻게 측정되는가: **len of characters**

```
# langchain 패키지에서 RecursiveCharacterTextSplitter 클래스 가져오기
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

- **RecursiveCharacterTextSplitter** 클래스 = 텍스트를 재귀적으로 분할 하는 기능 제공
 - **chunk_size** = 분할할 청크의 크기
 - **chunk_overlap** = 인접 청크 간의 겹침 크기
 - **length_function** = 청크의 길이를 계산하는 함수
 - **is_separator_regex** = 구분자가 정규 표현식인지 여부를 지정하는 매개변수
 - 예시: 청크 크기를 100, 겹침 크기를 20으로 설정하고, 길이 계산 함수로 **len** 을 사용하며, 구분자가 정규 표현식이 아님을 나타내기 위해 **is_separator_regex** 를 **False** 로 설정

```
recursive_text_splitter = RecursiveCharacterTextSplitter(
    # 정말 작은 청크 크기로 설정
    chunk_size=100,
    chunk_overlap=10,
    length_function=len,
    is_separator_regex=False,
)
```

```
# chain of density 논문의 일부 내용을 불러오기
with open("../12_RAG/data/chain-of-density.txt", "r") as f:
    text = f.read()[:500]
```

```
# character text splitter
character_text_splitter = CharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=10,
    separator=" "
)

for sent in character_text_splitter.split_text(text):
    print(sent)
print("===" * 20)

# recursive_text_splitter
recursive_text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=10
)

for sent in recursive_text_splitter.split_text(text):
    print(sent)
```

- **character text splitter**

Selecting the “right” amount of information to include in a summary is a difficult task.
A good
A good summary should be detailed and entity-centric without being overly dense and hard to follow.
to follow. To better understand this tradeoff, we solicit increasingly dense GPT-4 summaries with
with what we refer to as a “Chain of Density” (CoD) prompt. Specifically, GPT-4 generates an initial
an initial entity-sparse summary before iteratively incorporating missing salient entities without
without increasing the length. Summaries genera
=====

- **recursive_text_splitter**

Selecting the “right” amount of information to include in a summary is a difficult task.
A good summary should be detailed and entity-centric without being overly dense and hard to follow.
follow. To better understand this tradeoff, we solicit increasingly dense GPT-4 summaries with what
with what we refer to as a “Chain of Density” (CoD) prompt. Specifically, GPT-4 generates an
an initial entity-sparse summary before iteratively incorporating missing salient entities without
without increasing the length. Summaries genera

3) Semantic Similarity

- **의미적 유사성을 기준** → 텍스트 분할
 - 출처: [Greg Kamradt's Notebook](#)
 - 높은 수준 (high level)에서 문장으로 분할 → 3개의 문장으로 그룹화 → 임베딩 공간 → 유사한 문장을 병합 하는 방식

- 사전 VS Code 터미널에서 설치할 것

```
pip install -U langchain langchain_experimental -q
```

```
from langchain_experimental.text_splitter import SemanticChunker
from langchain_huggingface import HuggingFaceEmbeddings
```

```
# SemanticChunker 생성하기
semantic_text_splitter = SemanticChunker(
    embeddings,
    add_start_index=True)
```

```
# chain of density 논문의 일부 내용 불러오기
with open("data/chain-of-density.txt", "r") as f:
    text = f.read()

for sent in semantic_text_splitter.split_text(text):
    print(sent, "\n")
    print("\n", "===" * 20, "\n")
```

- Sementic Similarity (0.4s)

```
Selecting the “right” amount of information to include in a summary is a difficult task. A good summary should be
Automatic summarization has come a long way in the past few years, largely due to a paradigm shift away from supe

=====

A summary is uninformative if it contains insufficient detail. If it contains too much information, however, it c


=====
```

✓ 3. 단계 3: 임베딩 (Embedding)

✓ 1) 유료 과금 임베딩 (OpenAI)

```
from langchain_community.vectorstores import FAISS
from langchain_openai.embeddings import OpenAIEmbeddings
```

```
# 단계 3: 임베딩 & 벡터스토어 생성(Create Vectorstore)
# 벡터스토어 생성하기
vectorstore = FAISS.from_documents(
    documents=splits, embedding=OpenAIEmbeddings())
```

- OpenAI Embeddding 모델들의 목록
 - 기본값 = text-embedding-ada-002
 -  openai-embedding-model

```
vectorstore = FAISS.from_documents(
    documents=splits, embedding=OpenAIEmbeddings(model="text-embedding-3-small")
)
```

- Warning: model not found. Using cl100k_base encoding.

✓ 2) 무료 Open Source 기반 임베딩

- 사전에 VS Code 터미널에 설치할 것

```
pip install fastembed -q
```

```
from langchain_community.embeddings import HuggingFaceBgeEmbeddings

# 단계 3: 임베딩 & 벡터스토어 생성(Create Vectorstore)
# 벡터스토어 생성하기
vectorstore = FAISS.from_documents(
    documents=splits, embedding=HuggingFaceBgeEmbeddings()
)
```

- HuggingFaceBgeEmbeddings 설치 결과 (1m 32.6s)

```
✓ 11.3s
Fetching 5 files: 100% ██████████ 5/5 [00:07<00:00, 4.38s/it]
tokenizer.json: █████ 711k/? [00:00<00:00, 23.4MB/s]
special_tokens_map.json: 100% ██████████ 695/695 [00:00<00:00, 48.8kB/s]
tokenizer_config.json: █████ 1.24k/? [00:00<00:00, 81.5kB/s]
config.json: 100% ██████████ 706/706 [00:00<00:00, 14.9kB/s]
model_optimized.onnx: 100% ██████████ 66.5M/66.5M [00:06<00:00, 9.56MB/s]
```

```
from langchain_community.embeddings.fastembed import FastEmbedEmbeddings

vectorstore = FAISS.from_documents(
    documents=splits,
    embedding=FastEmbedEmbeddings()
)
```

- FastEmbedEmbeddings 설치 결과 (11.3s)

```
✓ 1m 32.6s
modules.json: 100% ██████████ 349/349 [00:00<00:00, 57.9kB/s]
config_sentence_transformers.json: 100% ██████████ 124/124 [00:00<00:00, 25.8kB/s]
README.md: █████ 90.3k/? [00:00<00:00, 7.98MB/s]
sentence_bert_config.json: 100% ██████████ 52.0/52.0 [00:00<00:00, 8.85kB/s]
config.json: 100% ██████████ 720/720 [00:00<00:00, 116kB/s]
model.safetensors: 100% ██████████ 1.34G/1.34G [01:19<00:00, 119MB/s]
tokenizer_config.json: 100% ██████████ 366/366 [00:00<00:00, 32.5kB/s]
vocab.txt: █████ 232k/? [00:00<00:00, 15.0MB/s]
tokenizer.json: █████ 711k/? [00:00<00:00, 29.4MB/s]
special_tokens_map.json: 100% ██████████ 125/125 [00:00<00:00, 6.25kB/s]
config.json: 100% ██████████ 191/191 [00:00<00:00, 31.7kB/s]
```

4. 단계 4: 벡터스토어 생성 (Create Vectorstore)

```
from langchain_community.vectorstores import FAISS

# FAISS DB 적용
vectorstore = FAISS.from_documents(
    documents=splits, embedding=embeddings
) # 0.3s
```

```
from langchain_community.vectorstores import Chroma

# Chroma DB 적용
vectorstore = Chroma.from_documents(
    documents=splits, embedding=embeddings
) # 2.3s
```

5. 단계 5: Retriever 생성

- **Retriever**
 - 구조화되지 않은 쿼리가 주어지면 문서를 반환하는 인터페이스
 - 문서를 저장할 필요 없이 문서를 반환(또는 검색)
 - 참고: [공식 문서](#)
 - 생성된 VectorStore에 `as_retriever()` 로 가져와서 **Retriever** 를 생성

✓ 1) 유사도 기반 검색

- 기본값 = **코사인 유사도** = **similarity** 적용됨

```
query = "회사의 저출생 정책이 뭐야?"

retriever = vectorstore.as_retriever(search_type="similarity")
search_result = retriever.get_relevant_documents(query)
print(search_result)
```

- 유사도 기반 검색 (**0.1s**)

```
[Document(metadata={'source': 'https://n.news.naver.com/article/437/0000378416'}, page_content="출산 직원에게 '1억원'
```

- **similarity_score_threshold** = 유사도 기반 검색에서 **score_threshold** 이상인 결과만 반환

```
query = "회사의 저출생 정책이 뭐야?"

retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"score_threshold": 0.4}
)

search_result = retriever.get_relevant_documents(query)

print(search_result)
```

- **score_threshold** = 0.4 로 조정
 - **score_threshold** = 0.8로 조정 시 나오지 않음

```
[Document(metadata={'source': 'https://n.news.naver.com/article/437/0000378416'}, page_content="출산 직원에게 '1"
```

- **MMR** (*maximum marginal search result*) 사용해보기

```
query = "회사의 저출생 정책이 뭐야?"

retriever = vectorstore.as_retriever(
    search_type="mmr", # 검색유형을 MMR로 바꾸기
    search_kwargs={"k": 2}
)

search_result = retriever.get_relevant_documents(query)

print(search_result)
```

- **MMR**

```
[Document(metadata={'source': 'https://n.news.naver.com/article/437/0000378416'}, page_content="출산 직원에게 '1억원'
```

✓ 2) 다양한 쿼리 생성

```
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain_google_genai import ChatGoogleGenerativeAI

import os
from dotenv import load_dotenv
```



```
load_dotenv()

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash-lite",
    temperature=0,
    max_output_tokens=4096,
)
# temperature = 0으로 설정
```

```
query = "회사의 저출생 정책이 뭐야?"

retriever_from_llm = MultiQueryRetriever.from_llm(
    retriever=vectorstore.as_retriever(), llm=gemini_lc
)
```

```
# Set logging for the queries
import logging

logging.basicConfig()
logging.getLogger("langchain.retrievers.multi_query").setLevel(logging.INFO)
```

```
print(type(unique_docs))          # <class 'list'>
```

```
unique_docs = retriever_from_llm.get_relevant_documents(query=question)
len(unique_docs)
print(f"문서의 수: {len(unique_docs)}")
```

- 문서의 수: 3 (1.3s)

```
INFO:langchain.retrievers.multi_query:Generated queries: ['부영그룹의 저출산 대책은 무엇인가요?', '부영그룹이 출산을 장려하기 위해']
```

3) Ensemble Retriever

```
from langchain.retrievers import BM25Retriever, EnsembleRetriever
from langchain_community.vectorstores import FAISS
```

```
doc_list = [
    "난 오늘 많이 먹어서 배가 정말 부르다",
    "떠나는 저 배가 오늘 마지막 배인가요?",
    "내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.",
]
```

```
# initialize the bm25 retriever and faiss retriever
bm25_retriever = BM25Retriever.from_texts(doc_list)
bm25_retriever.k = 2

faiss_vectorstore = FAISS.from_texts(doc_list, embeddings)
faiss_retriever = faiss_vectorstore.as_retriever(search_kwargs={"k": 2})

# initialize the ensemble retriever
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, faiss_retriever],
    weights=[0.5, 0.5]
)
```

```
def pretty_print(docs):
    for i, doc in enumerate(docs):
        print(f"[{i+1}] {doc.page_content}")
```

```
sample_query = "나 요즘 배에 정말 살이 많이 찼어..."
print(f"[Query]\n{sample_query}\n")
relevant_docs = bm25_retriever.get_relevant_documents(sample_query)
print("[BM25 Retriever]")
pretty_print(relevant_docs)
print("===" * 20)
relevant_docs = faiss_retriever.get_relevant_documents(sample_query)
print("[FAISS Retriever]")
pretty_print(relevant_docs)
print("===" * 20)
```

```
print("==== * 20")
relevant_docs = ensemble_retriever.get_relevant_documents(sample_query)
print("[Ensemble Retriever]")
pretty_print(relevant_docs)
```

- sample_query

```
[Query]
나 요즘 배에 정말 살이 많이 찼어...

[BM25 Retriever]
[1] 난 오늘 많이 먹어서 배가 정말 부르다
[2] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
=====

[FAISS Retriever]
[1] 떠나는 저 배가 오늘 마지막 배인가요?
[2] 난 오늘 많이 먹어서 배가 정말 부르다
=====

[Ensemble Retriever]
[1] 난 오늘 많이 먹어서 배가 정말 부르다
[2] 떠나는 저 배가 오늘 마지막 배인가요?
[3] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
```

```
sample_query = "바다 위에 떠다니는 배들이 많다"
print(f"[Query]\n{sample_query}\n")
relevant_docs = bm25_retriever.get_relevant_documents(sample_query)
print("[BM25 Retriever]")
pretty_print(relevant_docs)
print("==== * 20")
relevant_docs = faiss_retriever.get_relevant_documents(sample_query)
print("[FAISS Retriever]")
pretty_print(relevant_docs)
print("==== * 20")
relevant_docs = ensemble_retriever.get_relevant_documents(sample_query)
print("[Ensemble Retriever]")
pretty_print(relevant_docs)
```

- sample_query_2

```
[Query]
바다 위에 떠다니는 배들이 많다

[BM25 Retriever]
[1] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[2] 떠나는 저 배가 오늘 마지막 배인가요?
=====

[FAISS Retriever]
[1] 떠나는 저 배가 오늘 마지막 배인가요?
[2] 난 오늘 많이 먹어서 배가 정말 부르다
=====

[Ensemble Retriever]
[1] 떠나는 저 배가 오늘 마지막 배인가요?
[2] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[3] 난 오늘 많이 먹어서 배가 정말 부르다
```

```
sample_query = "ships"
print(f"[Query]\n{sample_query}\n")
relevant_docs = bm25_retriever.get_relevant_documents(sample_query)
print("[BM25 Retriever]")
pretty_print(relevant_docs)
print("==== * 20")
relevant_docs = faiss_retriever.get_relevant_documents(sample_query)
print("[FAISS Retriever]")
pretty_print(relevant_docs)
print("==== * 20")
relevant_docs = ensemble_retriever.get_relevant_documents(sample_query)
print("[Ensemble Retriever]")
pretty_print(relevant_docs)
```

- sample_query_3

```
[Query]
ships
```

```
[BM25 Retriever]
[1] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[2] 떠나는 저 배가 오늘 마지막 배인가요?
```

```
[FAISS Retriever]
[1] 난 오늘 많이 먹어서 배가 정말 부르다
[2] 떠나는 저 배가 오늘 마지막 배인가요?
```

```
[Ensemble Retriever]
[1] 떠나는 저 배가 오늘 마지막 배인가요?
[2] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[3] 난 오늘 많이 먹어서 배가 정말 부르다
```

```
sample_query = "pear"
print(f"[Query]\n{sample_query}\n")
relevant_docs = bm25_retriever.get_relevant_documents(sample_query)
print("[BM25 Retriever]")
pretty_print(relevant_docs)
print("==" * 20)
relevant_docs = faiss_retriever.get_relevant_documents(sample_query)
print("[FAISS Retriever]")
pretty_print(relevant_docs)
print("==" * 20)
relevant_docs = ensemble_retriever.get_relevant_documents(sample_query)
print("[Ensemble Retriever]")
pretty_print(relevant_docs)
```

- sample_query_4

```
[Query]
pear

[BM25 Retriever]
[1] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[2] 떠나는 저 배가 오늘 마지막 배인가요?
```

```
[FAISS Retriever]
[1] 난 오늘 많이 먹어서 배가 정말 부르다
[2] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
```

```
[Ensemble Retriever]
[1] 내가 제일 좋아하는 과일들은 배, 사과, 키위, 수박 입니다.
[2] 난 오늘 많이 먹어서 배가 정말 부르다
[3] 떠나는 저 배가 오늘 마지막 배인가요?
```

6. 단계 6: 프롬프트 생성 (Create Prompt)

- 프롬프트 엔지니어링 = 주어진 데이터(context)를 토대로 우리가 원하는 결과를 도출할 때 중요한 역할
 - tip_1
 - retriever에서 도출한 결과에서 중요한 정보가 누락된다면 → retriever의 로직을 수정해야 함
 - retriever에서 도출한 결과가 많은 정보를 포함하고 있지만, llm이 그 중에서 중요한 정보를 찾지 못하거나 원하는 형태로 출력하지 않는다면 → 프롬프트를 수정해야 함
 - tip_2
 - LangSmith의 hub에는 검증된 프롬프트가 많이 업로드 되어 있음
 - 검증된 프롬프트를 활용하거나 약간 수정한다면 비용과 시간을 절약할 수 있음
 - [LangSmith-hub-rag](#)

```
from langchain import hub

prompt = hub.pull("rlm/rag-prompt")
prompt
```

- langchain의 hub에서 받은 프롬프트

7. 단계 7: 언어모델 생성 (Create LLM)

- gemini 모델 사용해보기

```
import os
from dotenv import load_dotenv
from google import genai
```

- 토큰 사용량 확인해보기

```
# .env 파일에서 환경 변수 로드
load_dotenv()

# GOOGLE_API_KEY 확인 (로컬 환경 변수 사용)
if "GOOGLE_API_KEY" not in os.environ:
    raise ValueError("GOOGLE_API_KEY 환경 변수가 설정되지 않았습니다. .env 파일을 확인해 주세요.")

# 클라이언트 객체 초기화 (모듈 임포트 전에 처리)
GENAI_CLIENT = genai.Client(api_key=os.environ["GOOGLE_API_KEY"])

# utils 폴더에서 함수 임포트
from utils.local_gemini_billing import generate_and_track_tokens

print("✅ 초기화 및 함수 로드 성공")
```

```
# 사용자 쿼리
test_prompt = "대한민국의 수도는 어디인가요?"
```

```
# 초기화된 GENAI_CLIENT 객체를 함수의 첫 번째 인수로 전달합니다.
answer, tokens = generate_and_track_tokens(GENAI_CLIENT, test_prompt, "gemini-2.5-flash-lite")

print(f"답변: {answer}")

if tokens:
    print("\n-----")
    print("🌟 Gemini 모델 API 토큰 사용량 (과금 정보) 🌟")
    print(f"  - 전체 사용 토큰: {tokens['total_tokens']}")
    print("-----")
```

- 사용 모델: gemini-2.5-flash-lite
- 질문: 대한민국의 수도는 어디인가요?

- 과금 계산해보기 (gemini)

응답: 대한민국의 수도는 **서울**입니다.

```
-----
🌟 Gemini 모델 API 토큰 사용량 (과금 정보) 🌟
- 입력(프롬프트) 토큰: 10
- 출력(응답) 토큰: 10
- 전체 사용 토큰: 20
-----
```

- HuggingFace 에 업로드 되어 있는 오픈소스 모델

- 참고: [HuggingFace LLM Leaderboard](#)

```
# HuggingFaceHub 객체 생성
from langchain.llms import HuggingFaceHub

repo_id = "google/flan-t5-xxl"

t5_model = HuggingFaceHub(
    repo_id=repo_id, model_kwargs={"temperature": 0.1, "max_length": 512}
)
```

```
t5_model.invoke("Where is the capital of South Korea?")
```

- 'seoul'

1) RAG 템플릿 실험

```
# 단계 1: 문서 로드(Load Documents)
# 문서를 로드, 청크 나누기, 인덱싱
from langchain.document_loaders import PyPDFLoader

# PDF 파일 로드. 파일의 경로 입력
file_path = "../12_RAG/data/SPRI_AI_Brief_2023년12월호_F.pdf"
loader = PyPDFLoader(file_path=file_path)

# 단계 2: 문서 분할(Split Documents)
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50)

split_docs = loader.load_and_split(text_splitter=text_splitter)          # 1.9s
```

```
# 단계 3, 4: 임베딩 & 벡터스토어 생성(Create Vectorstore)
# 벡터스토어 생성하기
vectorstore = FAISS.from_documents(documents=split_docs, embedding=embeddings)

# 단계 5: 리트리버 생성(Create Retriever)
# 사용자의 질문(query) 에 부합하는 문서를 검색하기
retriever = vectorstore.as_retriever(search_type="similarity")
```

- 유사도 기반 retriever

```
[Document(id='a3c05b85-9b92-4a13-be69-475b5e4a4feb', metadata={'source': 'https://n.news.naver.com/article/437/00
```

```
# 유사도 높은 K 개의 문서를 검색하기
k = 3

# (Sparse) bm25 retriever and (Dense) faiss retriever 를 초기화하기
bm25_retriever = BM25Retriever.from_documents(split_docs)
bm25_retriever.k = k

faiss_vectorstore = FAISS.from_documents(split_docs, embeddings)
faiss_retriever = faiss_vectorstore.as_retriever(search_kwargs={"k": k})

# initialize the ensemble retriever
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, faiss_retriever], weights=[0.5, 0.5]
)

# 1.8s
```

```
# 단계 6: 프롬프트 생성(Create Prompt)
# 프롬프트 생성하기
prompt = hub.pull("rlm/rag-prompt")

print(prompt)
```

- rag-prompt

```
input_variables=['context', 'question'] input_types={} partial_variables={} metadata={'lc_hub_owner': 'rlm', 'lc_
```

```
# 단계 7: 언어모델 생성(Create LLM)
# LLM 생성하기
from google.genai.errors import APIError
from langchain_google_genai import ChatGoogleGenerativeAI

import os
from dotenv import load_dotenv

load_dotenv()

# API 키 확인
if not os.getenv("GOOGLE_API_KEY"):
    os.environ["GOOGLE_API_KEY"] = input("Enter your Google API key: ")
```

```
# LLM 초기화
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,
    max_output_tokens=4096,
)

# temperature = 0으로 설정

def format_docs(docs):
    # 검색한 문서 결과를 하나의 문단으로 합치기
    return "\n\n".join(doc.page_content for doc in docs)
```

```
# 단계 8: 체인 생성(Create Chain)
rag_chain = (
    {"context": ensemble_retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

# 결과 출력
print(f"PDF Path: {file_path}")
print(f"문서의 수: {len(docs)}")
print("==== * 20")
print(f"[HUMAN]\n{question}\n")
print(f"[AI]\n{response}")
```

• query_1

PDF Path: data/SPRI_AI_Brief_2023년12월호_F.pdf

문서의 수: 23

=====

[HUMAN]

삼성 가우스에 대해 설명해주세요

[AI]

삼성 가우스는 삼성전자가 개발한 생성 AI 모델로, 언어, 코드, 이미지의 3개 모델로 구성되어 있습니다. 이 모델은 온디바이스에서 작동 가능하며, 외부

```
# 단계 8: 체인 실행(Run Chain)
# 문서에 대한 질의를 입력하고, 답변을 출력하기
question = "삼성 가우스에 대해 설명해주세요"
response = rag_chain.invoke(question)
print(response)
```

• query_2 (4.7s)

삼성 가우스는 삼성전자가 자체 개발한 생성형 AI 모델로, 2023년 11월 '삼성 AI 포럼'에서 공개되었습니다. 이 모델은 언어, 코드, 이미지의 세 가지 도

```
# 단계 8: 체인 실행(Run Chain)
# 문서에 대한 질의를 입력하고, 답변을 출력하기
question = "미래의 AI 소프트웨어 매출 전망은 어떻게 되나요?"
response = rag_chain.invoke(question)
print(response)
```

• query_3 (7.2s)

IDC는 AI 소프트웨어 시장이 2027년에 2,510억 달러에 달할 것으로 전망합니다. 이는 2022년 640억 달러에서 연평균 31.4% 성장한 수치입니다. 특히

```
# 단계 8: 체인 실행(Run Chain)
# 문서에 대한 질의를 입력하고, 답변을 출력하기
question = "YouTube 가 2024년에 의무화 한 것은 무엇인가요?"
response = rag_chain.invoke(question)
print(response)
```

• query_4 (3.8s)

YouTube는 2024년부터 생성 AI를 사용한 콘텐츠에 AI 라벨 표시를 의무화했습니다. 이를 준수하지 않을 경우 콘텐츠가 삭제되거나 크리에이터에 대한 수의

