EP01. #RAG의 동작 과정 쉽게 이해하기

(1) LLM vs. RAG

- LLM : 외부 정보 참고 X
- LLM **+ RAG**: 외부 정보 참고 O
    - 외부 정보 = 참고 정보 (context)

(2) Chunking

참고 정보 예시: pdf 문서

통째로 LLM의 입력에 넣어줄 경우의 문제점:

- (1) Too costly
- (2) Lost in the middle

해결책: **Chunking**

- Chunk 예시: 1개의 단락 (1000 token)
- 주로 overlap (O)

(3) Pre-processing 단계 요약

- Step 1) Document load
- Step 2) Text split
- Step 3) Embedding
- Step 4) Store => Vector DB

EP02. #RAG의 동작 과정 쉽게 이해하기 (실행 단계)

Runtime 단계

(참고: 현재 Vector DB는 이미 준비 된 상태)

(1) 유사도 검색 (= Similarity search)

- User가 질문 시, retriever (=검색기)가 질문과의 연관도 (유사도) 계산

- 관련된 단락을 Vector DB에서 (Top K개) 뽑아낸 뒤, prompt에 넣어줌
- 결국, LLM의 입력에는 (1) + (2)가 함께 들어가게 됨
    - (1) User의 질문
    - (2) [Prompt] RAG를 통해 찾아온 정보 (Top K개의 chucnk)

(2) Retriever (검색기)의 필요성

- (1) 정확한 정보 제공
    - 질문과 가장 유사도(관련성) 높은 정보를 검색함
- (2) 응답 시간 단축
    - 효율적인 검색 알고리즘을 통해 관련있는 정보를 빠르게 검색
    - User experience에 매우 중요
- (3) 최적화
    - 필요하한 정보만을 추출 (불필요 데이터 처리 줄임)

(3) 동작 방식

- Step 1) 질문의 벡터화
- Step 2) 벡터 유사도 계산 & 비교
- Step 3) Top K 문서 선정
- Step 4) Prompt로 전달

(4) Retriever (검색기)의 종류

a) Sparse retriever

- User의 질문을 **discrete (keyword) vector**로 변환
- How? e.g., TF-IDF, BM25 (전통적인 방법들)
- 장/단점
    - 장점) 간단함/단어의 유무만을 가지고 판단
    - 단점) 의미적 연관성 고려 X

b) Dense retriever

- User의 질문을 **continuous vector**로 변환
- How? DL models

- 장/단점: Sparse retriever와 반대

EP03. PDF 문서 기반 QA (RAG 구축하기)

Procedure

- Step 1) Load document
- Step 2) Split (to chunks)
- Step 3) Embedding
- Step 4) VectorStore
- Step 5) Retrieval
- Step 6) Prompt
- Step 7) LLM
- Step 8) Output

Environment Setting

```
# 1-1) Load API key
from dotenv import load_dotenv
load_dotenv()

# 1-2) LangSmith (for tracking)
!pip install -qU langchain-teddynote
from langchain_teddynote import logging

# 1-3) Project Name
project_nm = "CH12-RAG"
logging.langsmith(project_nm)

# 1-4) Langchain packages
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PyMuPDFLoader
from langchain_community.vectorstores import FAISS
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
```

Step 1) Load document

```
import os
PATH = 'data'
fn = 'SPRI_AI_Brief_2023년12월호_F.pdf'
loader = PyMuPDFLoader(os.path.join(PATH,fn))
docs = loader.load()

print(f"# pages: {len(docs)}")
print(docs[10].page_content) # contents of 10th page
print(docs[10].__dict__) # meta data
```

Step 2) Split (to chunks)

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
split_documents = text_splitter.split_documents(docs)

print(f"# chunks: {len(split_documents)}")
print(split_documents[7].page_content) # content of 7th chunck
```

Step 3) Embedding

```
embeddings = OpenAIEmbeddings() # OpenAI
```

Step 4) VectorStore

```
# Create Vector DB (e.g, FAISS)
vectorstore = FAISS.from_documents(documents=split_documents, embedding=embeddings)

# ex) Find chunks from DB with high similarity
example_keyword = "구글"
for doc in vectorstore.similarity_search(example_keyword):
    print(doc.page_content)
```

Step 5) Retrieval

```
# Retreiver to search for chunks
retriever = vectorstore.as_retriever()

# Ex) Find the closests chunks
example_query = "삼성전자가 자체 개발한 AI 의 이름은?"
retriever.invoke(example_query)
```

Step 6) Prompt

```
# Create prompt
# Requirements: context & question
prompt = PromptTemplate.from_template(
    """You are an assistant for question-answering tasks.
Use the following pieces of retrieved context to answer the question.
If you don't know the answer, just say that you don't know.
Answer in Korean.

#Context:
{context}

#Question:
{question}

#Answer:"""
)
```

Step 7) LLM

```
# GPT-4o
llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
```

Step 8) Output

```
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

response = chain.invoke(example_query)
print(response)
```