LangChain

강사 : 한기영

Gooroomee

과정 개요

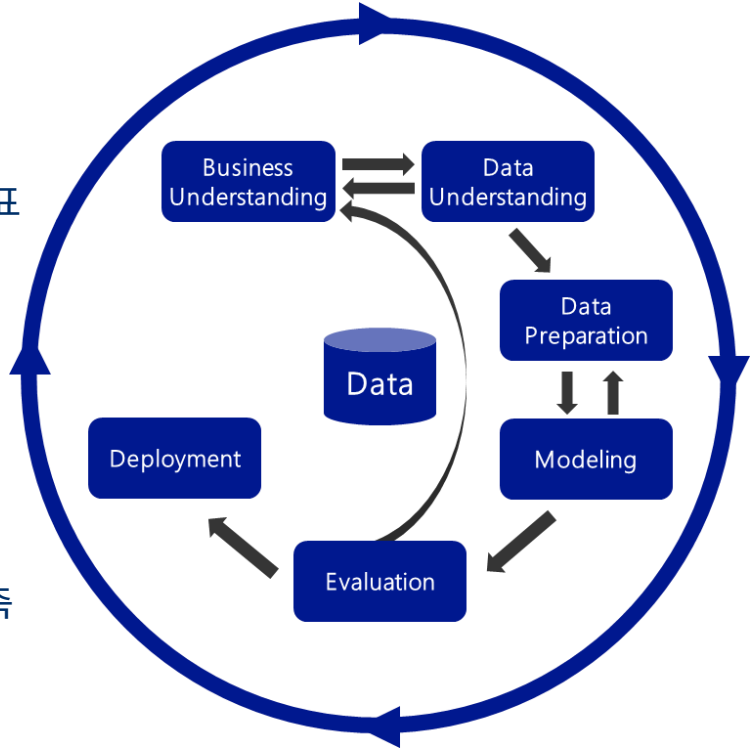
- Chapter 1. 개발환경 세팅
- Chapter 2. 언어모델 이해
- Chapter 3. LangChain : Model I/O
- Chapter 4. LangChain : RAG-VectorDB
- Chapter 5. LangChain : RAG-Memory

전체 Process(CRISP-DM)

무엇이 문제인가?

✓ 비즈니스 **문제정의**
✓ 데이터분석 방향, 목표
✓ 초기 **가설 수립**

$x \rightarrow y$



✓ 원본식별
✓ **분석을 위한 구조 만들기**
✓ **데이터분석 EDA & CDA**

✓ 모델링을 위한
데이터 구조 만들기
- 모든 셀은 **값**이 있어야 한다.
- 모든 값은 **숫자** 여야 한다.
- (필요 시) 숫자의 **범위**가 일치

✓ 모델 관리
✓ AI 서비스 구축

✓ 모델을 만들고
검증하기

문제가 해결 되었는가?

✓ 기술적 관점 평가
✓ 비즈니스 관점 평가

1. 개발환경 세팅

라이브러리 설치

✓ ① 구글 드라이브에 새 폴더 생성

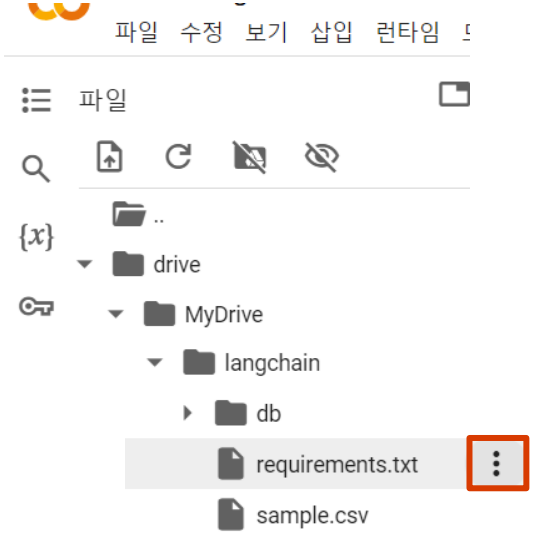
- 새 폴더 이름 : langchain

✓ ② Colab에서 구글 드라이브 연결

- Colab 코드에서 실행하여 연결

✓ ③ requirements.txt 파일 경로 확인

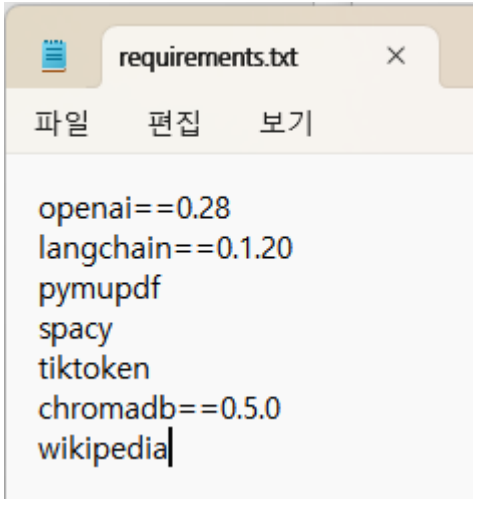
- 오른쪽 ⋮ 을 눌러, [경로 복사] 클릭



라이브러리 설치

✓ requirements.txt

- 설치할 라이브러리들을 한꺼번에 설치하도록 하는 파일
- 파일 이름은 변경될 수 있으나, 보통 requirements.txt 를 사용




```
# 경로 : /content/drive/MyDrive/langchain/requirements.txt
# 경로가 다른 경우 아래 코드의 경로 부분을 수정하세요.

!pip install -r
/content/drive/MyDrive/langchain/requirements.txt
```

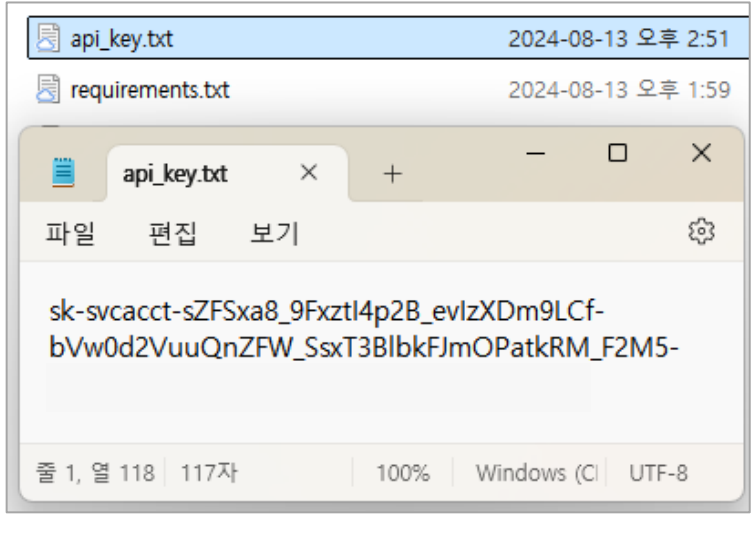
OpenAI API Key 설정

✓ 개인별 api key 제공

- 4명이 하나의 api key 사용
- 수업 중에만 키 사용
- ⚠️ 절대 api key가 github 등 인터넷 공간에 노출되지 않도록 해주세요.

✓ api_key.txt 파일 생성

- 새 텍스트 파일을 생성하여
- 내용 : openai api key를 입력
- 파일 이름 : api_key.txt
- 구글 드라이브 langchain 폴더에 업로드



OpenAI API Key 설정

✓ 코드에서 api key 로딩하고 설정하기

- 구글 드라이브 폴더를 통해
- api_key.txt 파일의 key 값을 읽어서
- openai.api_key 에 읽어 들인 key 지정

```python
def load_api_key(filepath):
    with open(filepath, 'r') as file:
        return file.readline().strip()

path = '/content/drive/MyDrive/langchain/'

# API 키 로드 및 설정
openai.api_key = load_api_key(path + 'api_key.txt')
```

⚠ 아래 코드셀은, 실행해서 key가 제대로 보이는지 확인하고 삭제하세요.

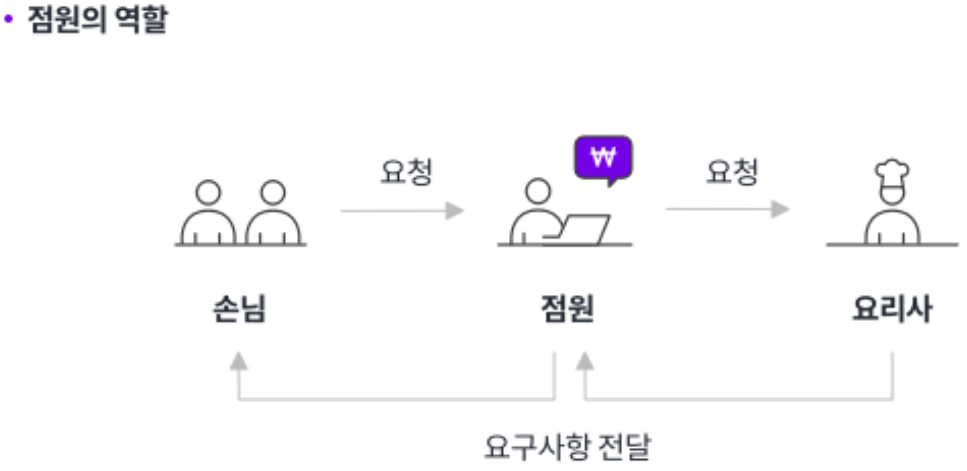
```python
print(openai.api_key)
```

API란?

✓ 예를 들어,

- 손님이 **점원**에게 주문을 하면
- **점원**은 요리사에게 주문을 전달
- 요리사는 요리를 만들어서 **점원**에게 전달
- **점원**은 요리를 손님에게 가져다 줍니다.

✓ 여기서 점원이 바로 API 입니다.

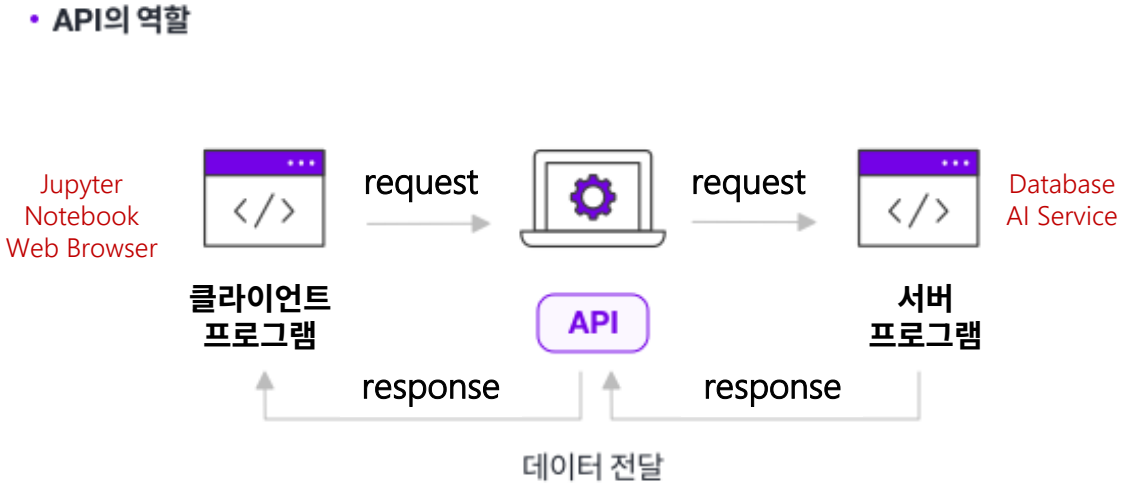


출처 : https://blog.wishket.com/api란-쉽게-설명-그린클라이언트/

API란?

✓ API는

- 클라이언트 프로그램에게 요청을 받아
- 서버로 전달, 서버는 요청을 처리한 후
- 결과 데이터를 API에 전달
- API가 다시 데이터를 클라이언트로 전달



출처 : https://blog.wishket.com/api란-쉽게-설명-그린클라이언트/

소프트웨어 간에 데이터를 교환하고
통신할 수 있도록 하는 인터페이스를 제공

API란?

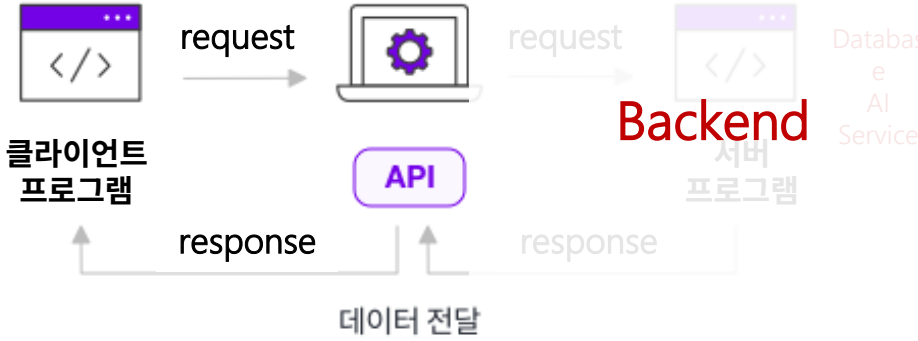
✓ API를 사용하기 위해서

- API의 뒷부분(Backend)은 잘 몰라도 Request와 Response만 알면 사용 가능

- **Request**
  - API 주소 + **API Key**
  - Request 형식 : 요청 양식

- **Response**
  - Response 형식 : 결과 양식

• **API의 역할**



출처 : https://blog.wishket.com/api란-쉽게-설명-그린클라이언트/

2. 언어 모델 이해

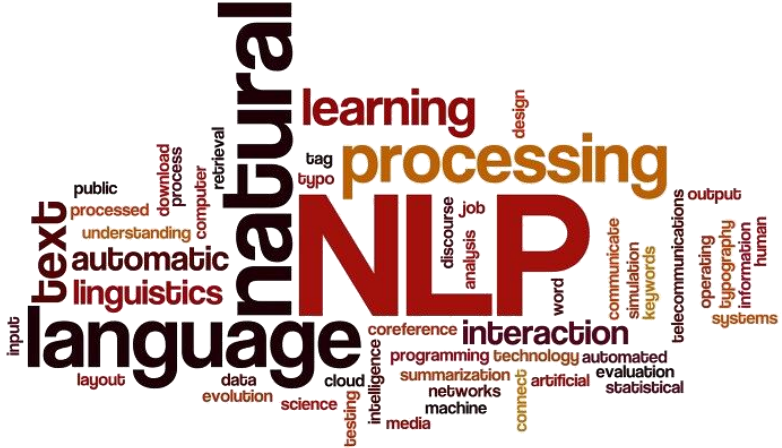
NLP Natural Language Processing

✓ NLP란?

- NLP는 인간 언어와 관련된 모든 것을 이해하는 데 초점을 맞춘 언어학 및 기계 학습 분야
- NLP 작업의 목표는 단일 단어를 개별적으로 이해하는 것뿐만 아니라 **해당 단어의 맥락을 이해**하는 것

✓ 일반적인 NLP 작업

- 문장 분류 : 리뷰의 감정 파악, 이메일 스팸 여부 감지 등
- 개체 명(사람, 위치, 조직 등) 인식
- 문장 생성
- 질문에 대한 답변
- 텍스트 번역, 요약



다음에 올 글자는?

“인공지능은 생각보다 __ __ __.”

Transformer

✓ 기존의 NLP : RNN 기반

- 오랫동안 언어모델을 위한 주요한 접근 방식
- 단점 : 병렬 처리 어려움, 장기 의존성 문제, 확장성 제한

✓ Transformer 등장 (Google, 2017, Attention Is All You Need, https://arxiv.org/abs/1706.03762)

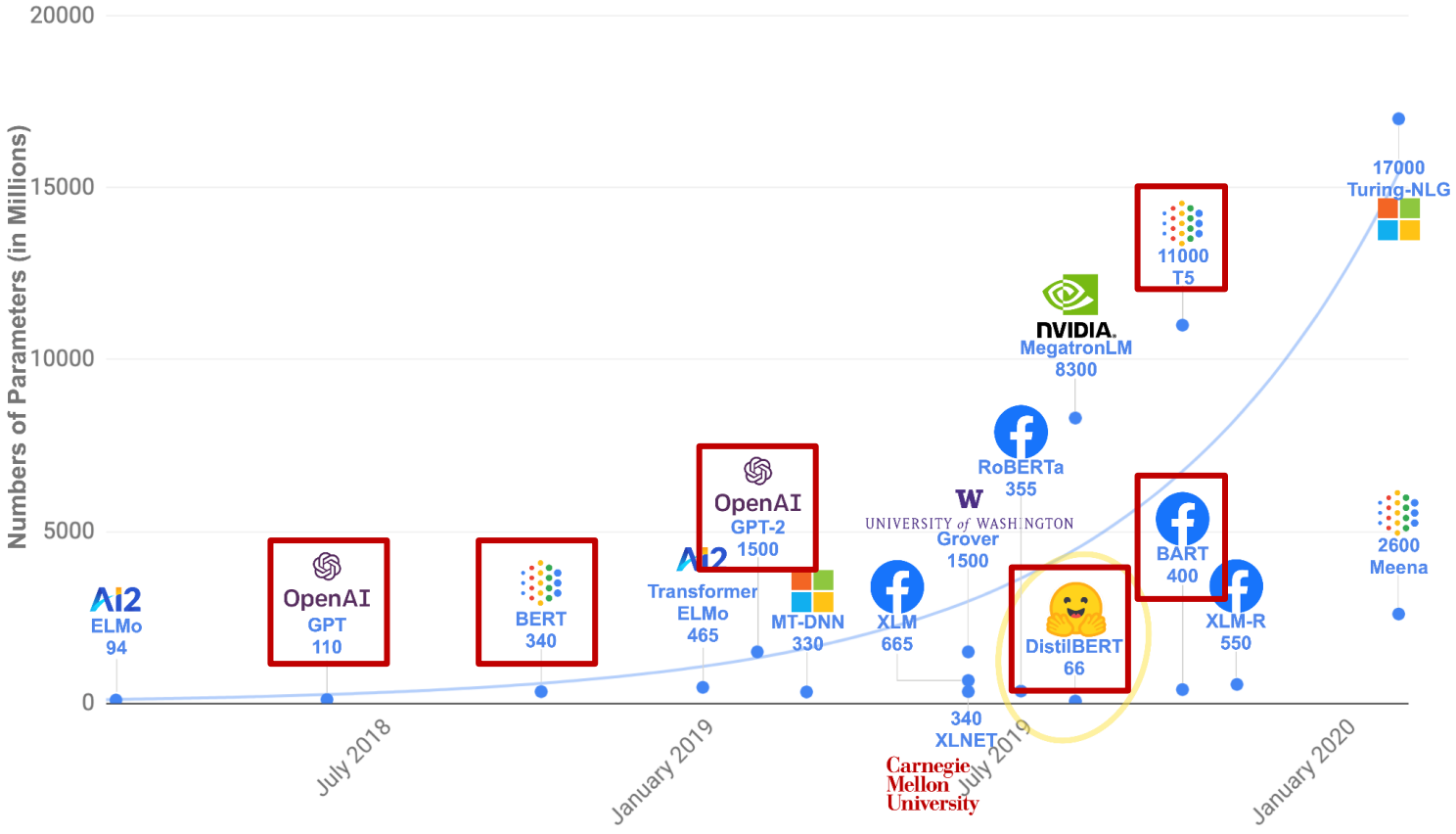
- RNN 모델의 단점을 극복
- 언어 모델의 Game Changer
- Transformer 덕분에 LLM이 발전하게 됨.

"트랜스포머가 NLP 세계를 완전히 휩쓸었다!" (2018)

Andrew Ng

Transformer

✓ 짧은 역사



Transformer

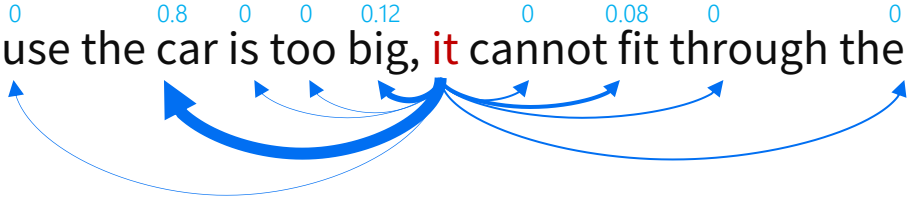
✓ Transformer의 특징

- 이전 문장들을 잘 기억
- 문맥상 **집중**해야 할 단어를 잘 캐치
  - 문장이나 단어 사이의 관계(문맥, context)를 파악하는 데 탁월한 능력

Attention

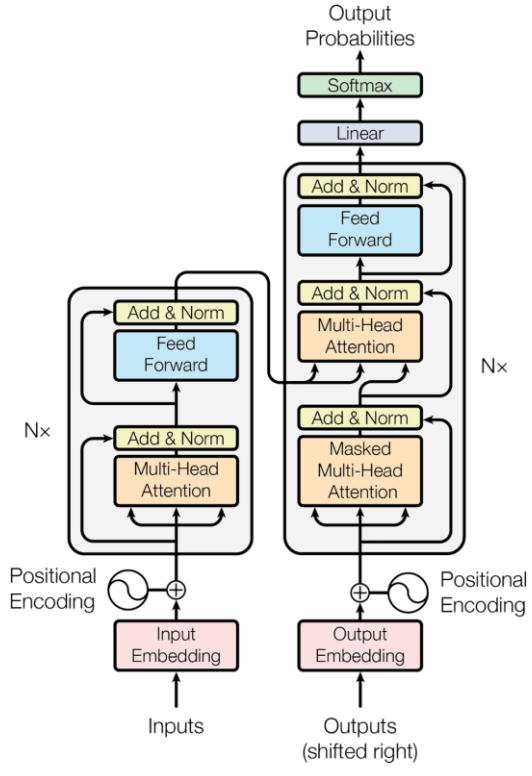
✓ 아래 문장에서 it 은 무엇을 가리킬까요?

- Because the car is too big, it cannot fit through the doorway.



attention score

Transformer

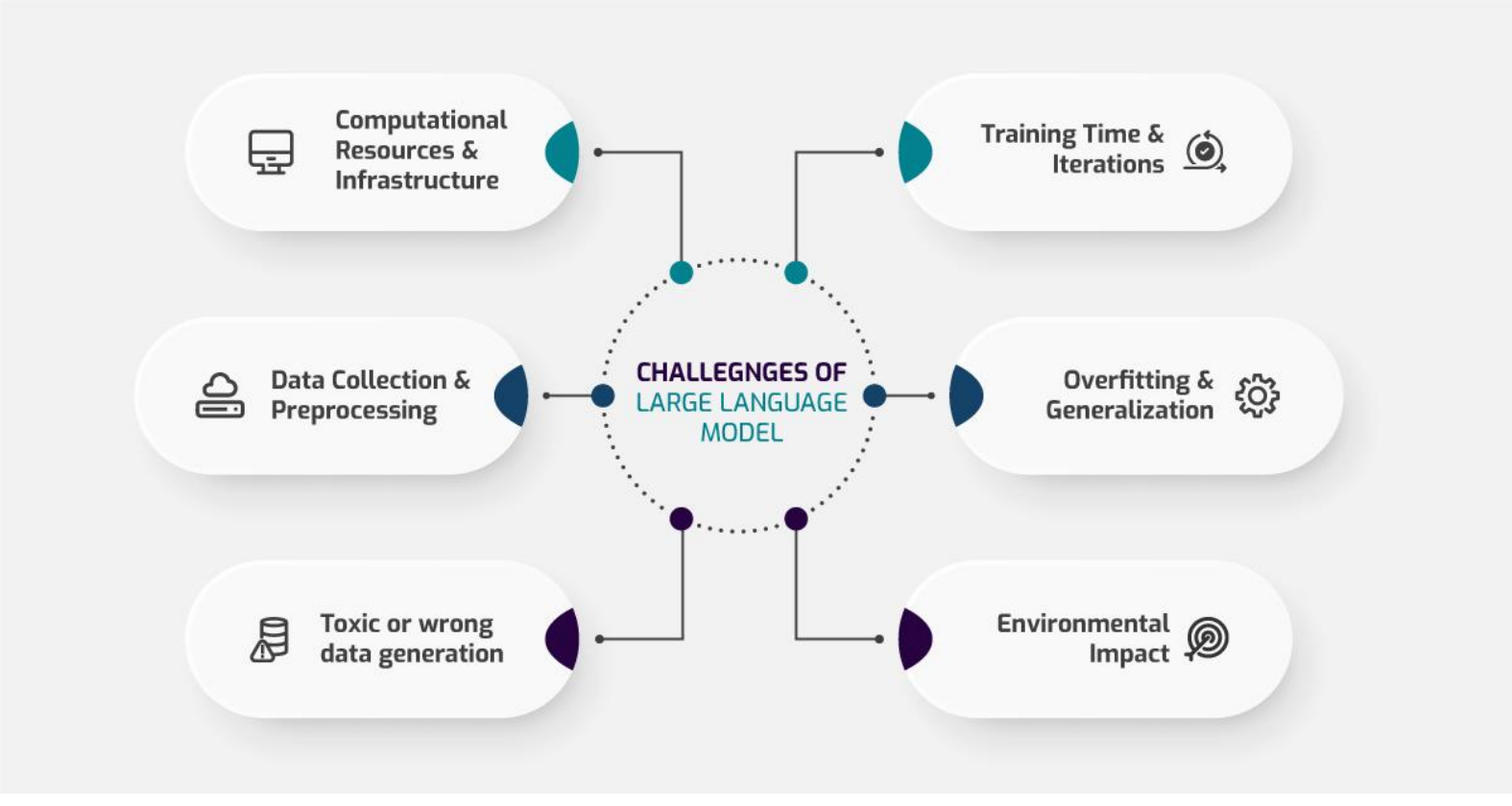
Studying	*Actual Work*
	`import transformers`

GPT, BERT

✓ Transformer를 기반으로 나온 언어 모델 : GPT, BERT

	학습 방식	특징
GPT **G**enerative **P**re-trained **T**ransformer	지금 배우고 있는 언어 모델은 진짜 ___.	생성적인 태스크에서 강점을 보임 (예: 텍스트 생성)
BERT **B**idirectional **E**ncoder **R**epresentations from **T**ransformers	랭체인을 이용하면 ____을 손쉽게 다루고 구현할 수 있습니다.	문맥을 양방향으로 이해할 수 있어, 문장 이해 및 의미 파악에 강점을 보임

LLM 모델 학습의 어려움



reference : https://www.shiksha.com/online-courses/articles/challenges-of-training-large-language-models-an-in-depth-look/

예제 파일

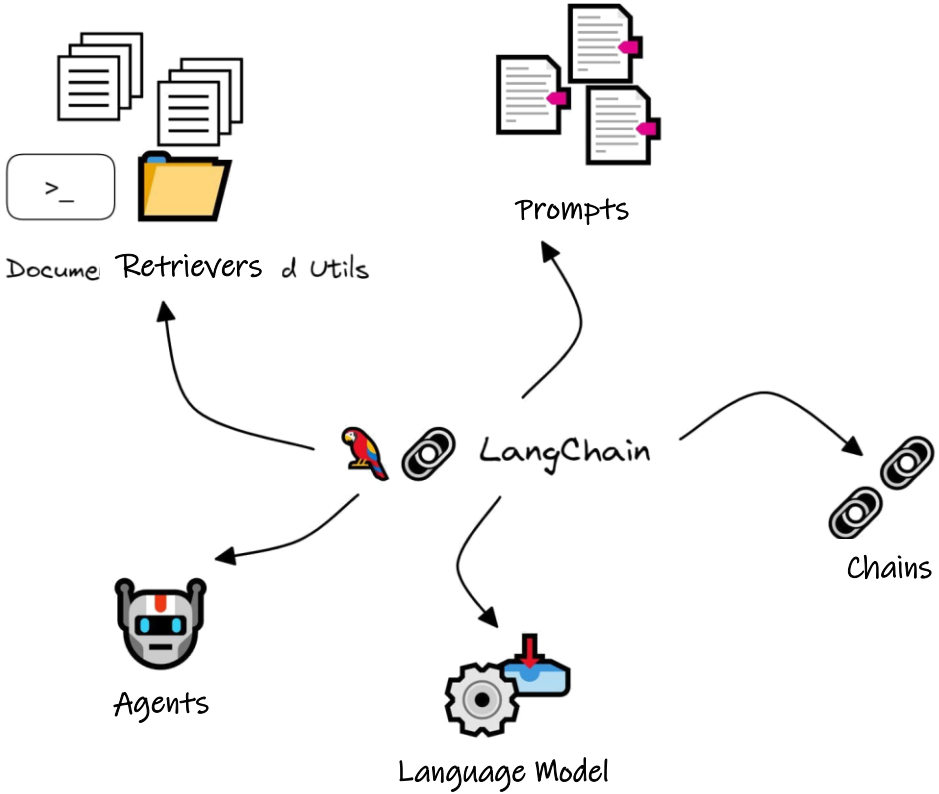
✓ 01. 일단 실행해보기.ipynb 파일을 열어서 예제를 실행해 봅시다.

3. LangChain① : Model I/O

LangChain

✓ LangChain이란?

- LangChain은 대규모 언어 모델(LLMs)을 활용하여 체인을 구성
- 이 체인을 통해, 복잡한 작업을 자동화하고 **쉽게** 수행할 수 있도록 돕는 라이브러리



https://wearefram.com/blog/langchain-101-build-your-own-gpt-powered-applications/

모델 사용

✓ OpenAI에서 제공하는 모델 사용

- LangChain에서 hugging Face의 모델 연결을 지원하지만, 이번 과정에서는 OpenAI 모델을 연결해서 사용합니다.
- LLM : **gpt-3.5-turbo**

```
from langchain.chat_models import ChatOpenAI
chat = ChatOpenAI(model = "gpt-3.5-turbo")
```

- 가격

모델	입력 토큰 100만 개	출력 토큰 100만 개
GPT-4o	$5	$15
GPT-4 Turbo	$10	$30
GPT-3.5 Turbo	$0.50	$1.50

모델 사용

✓ Langchain에서 사용하는 세 가지 유형의 Message

- **System**Message : 시스템 역할 부여
- **Human**Message : 질문
- **AI**Message : 답변

```
sys_role = '당신은 애국심을 가지고 있는 건전한 대한민국 국민입니다.'
question = "독도는 어느 나라 땅이야?"
                                  HumanMessage                                SystemMessage
result = chat([HumanMessage(content = question), SystemMessage(content = sys_role)])
          AIMessage
print(result.content)
```

모델 사용

✓ Message를 구분하는 이유

- **대화 구조화**
  - 각 메시지의 유형에 따라, 대화의 특정 부분을 정의하고, 시스템과 사용자가 어떻게 상호작용할지를 명확히 함.
- **역할 분리**
  - 메시지 유형을 구분함으로써 대화의 흐름을 관리하고, 특정 메시지가 언제, 어떻게 사용되는지를 쉽게 이해할 수 있음.
- **대화 관리**
  - 이러한 구조는 대화를 더 잘 이해하고 관리하는 데 도움을 줌.

모델 사용 – 입력 구조화

✓ PromptTemplate

- 파이썬 입력과 프롬프트를 결합하도록 지원하는 모듈
- template = "**{nation}**의 인구수는?"
  - {nation} : 입력 받을 변수
- Input_variables = ["**nation**"]
  - 입력 변수를 리스트로 선언

```
from langchain import PromptTemplate

prompt = PromptTemplate(template = "{nation}의 인구수는?",
                        input_variables = ["nation"])
```

- 사용
  - prompt.format( )

```
print(prompt.format(nation = "한국"))
print(prompt.format(nation = "영국"))
```

- 언어 모델과 결합

```
result = chat([HumanMessage(content=prompt.format(nation="한국"))])
print(result.content)
```

모델 사용 – 출력 구조화

✓ OutputParser

- 출력 형태를 지정하는 방법
- CommaSeparatedListOutputParser
  - 출력 형태 : 콤마로 구분된 리스트 형태

```
# 출력파서 선언
output_parser = CommaSeparatedListOutputParser()

# 사용
result = chat([HumanMessage(content = "트랜스포머 기반 언어모델 3개 알려줘."),
               HumanMessage(content = output_parser.get_format_instructions())])
                                                    # 모델에 지시사항 추가

# 결과 출력
output = output_parser.parse(result.content)
print(output)
```

모델 사용 – 출력 구조화

✓ 스트리밍 방식 출력

- 모델의 출력을 완성될 때까지 기다리지 않고, 생성되는 즉시 실시간으로 받을 수 있음.
- **streaming = True**
  - 모델의 응답을 스트리밍 방식으로 받아오겠다는 것을 의미
- **callbacks = [StreamingStdOutCallbackHandler()]**
  - 스트리밍 중에 생성된 응답을 실시간으로 콘솔 출력(표준 출력)으로 내보내는 콜백 핸들러

```
# 스트리밍 방식으로 출력
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler

chat2 = ChatOpenAI(model="gpt-3.5-turbo",
                   streaming=True, callbacks=[StreamingStdOutCallbackHandler()])
                   # 스트리밍 설정

sys_role = "당신은 세계 최고의 요리사입니다."
question = "한국식 후라이드치킨 만드는 방법 알려주세요."
result = chat2([HumanMessage(content=question), SystemMessage(content = sys_role)])
```

4. LangChain② : RAG - VectorDB

RAG Retrieval Augmented Generation

✓ 나에게 필요한 답변을 해주는 LLM을 만들려면...

-	나의 데이터를 가지고 직접 학습 시킨다.	Modeling
사전 학습 된 똘똘한 LLM	나의 데이터를 가지고 추가 학습 시킨다.	Fine-tuning
	나의 데이터를 가지고 답변 시킨다.	RAG

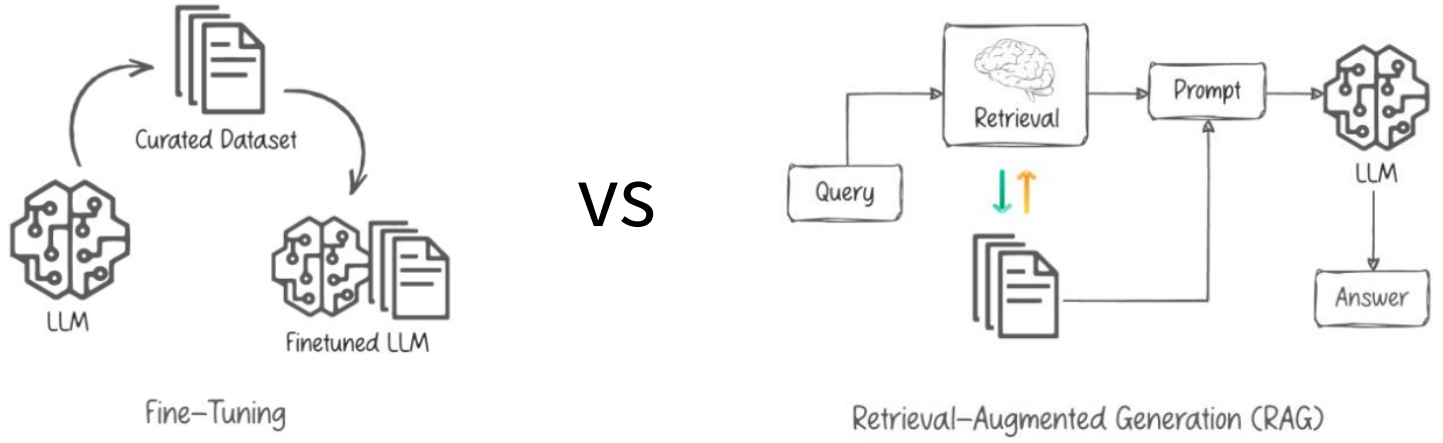


그림 출처 : https://www.linkedin.com/pulse/power-your-enterprise-llm-fine-tuning-vs-retrieval-augmentation-pgvif/

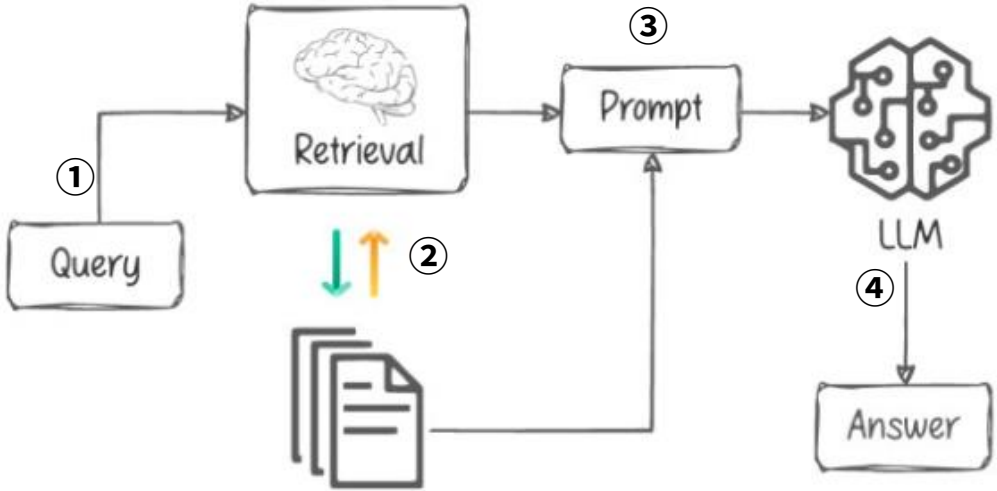
RAG Retrieval Augmented Generation

✓ LLM 모델

- 학습하지 않은 내용은 모름.

✓ LLM with RAG

- ① 사용자 질문을 받음
- ② **지식DB**에서 답변에 필요한 문서 검색
- ③ 필요한 문서를 포함한 프롬프트 생성
- ④ LLM이 답변 생성하기



Retrieval-Augmented Generation (RAG)

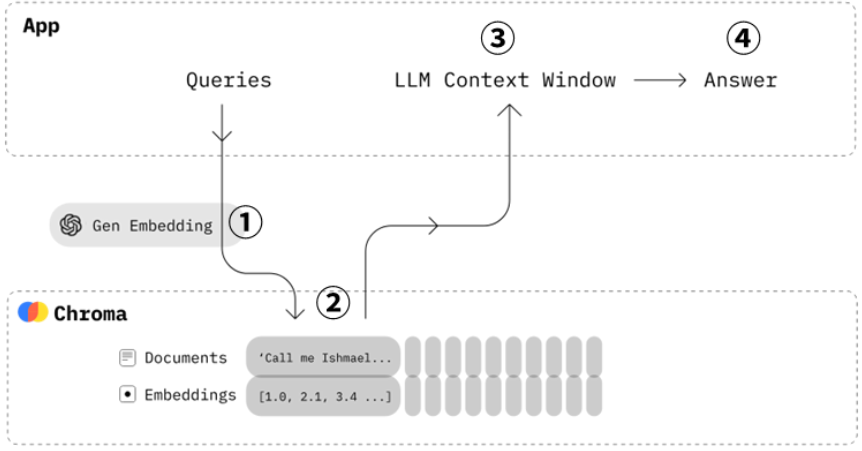
그림 출처 : https://www.linkedin.com/pulse/power-your-enterprise-llm-fine-tuning-vs-retrieval-augmentation-pgvif/

Vector DB

✓ 대규모 텍스트 데이터 및 임베딩 벡터를 저장, 검색용

✓ **LLM with RAG** 절차 다시 살펴보기

- ① 사용자 질문을 받음
  - 임베딩 : 벡터로 변환(**질문 벡터**)
- ② **준비된 정보 DB**에서 답변에 필요한 문서 검색
  - [**질문 벡터**]와 DB내 저장된 [**문서 벡터**]와 **유사도** 계산
  - 가장 유사도가 높은 문서 n개 찾기
- ③ 필요한 문서를 포함한 프롬프트 생성
- ④ LLM이 답변 생성하기



✓ **Vector DB로 Chroma DB 사용**

그림 출처 : https://docs.trychroma.com/

Vector DB

✓ Vector DB 구축 절차(Chroma DB)

- **텍스트 추출** : Document Loader
  - 다양한 문서(word, pdf, web page 등)로 부터 텍스트 추출하기
- **텍스트 분할** : Text Splitter
  - chunk 단위로 분할
  - Document 객체로 만들기
- **텍스트 벡터화** : Text Embedding
- **Vector DB로 저장** : Vector Store

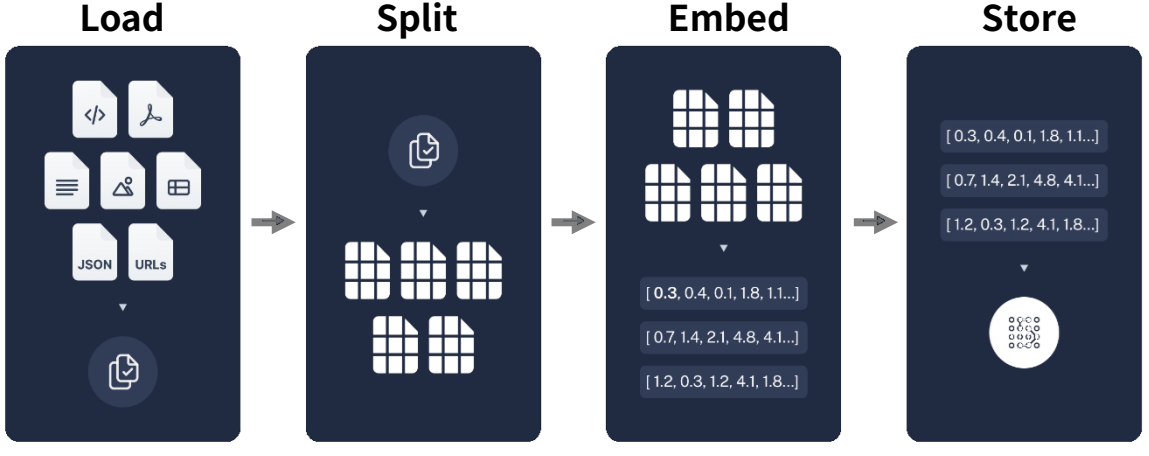


그림 출처 : https://python.langchain.com/v0.1/docs/use_cases/question_answering/

✓ Chroma DB

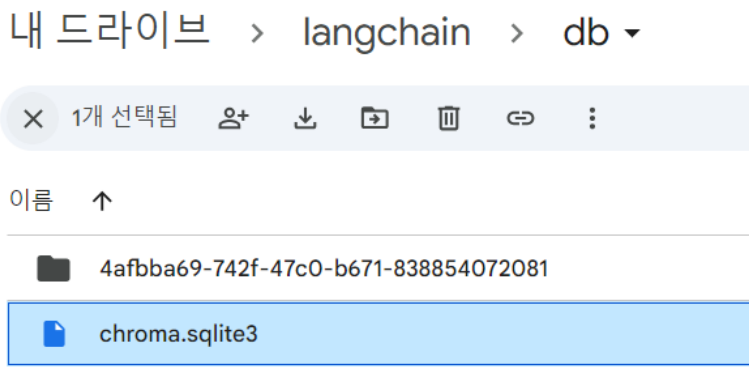
- SQLite3 기반 Vector DB
- DB Browser for SQLite3로 접속 가능

Vector DB

✓ DB 구성

- DB 경로 지정
  - 없으면, 새 폴더를 만들며 DB 생성
  - 있으면, 기존 DB 연결
- 임베딩 모델 지정

```
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

database = Chroma(persist_directory = path + "db",
                  embedding_function = embeddings)
```



Vector DB

✓ Insert 방법

- (1) 단순 텍스트 입력 : 각 단위 텍스트를 리스트 형태로 입력 **.add_texts()**

```
input_list1 = ['test 데이터 입력1', 'test 데이터 입력2']

# 입력 시 인덱스 저장(조회시 사용)
ind = database.add_texts(input_list1)
```

- (2) 텍스트와 메타데이터 입력 : 각 단위 텍스트와 메타정보를 함께 입력 **.add_documents()**

```
input_list2 = ['오늘 날씨는 매우 맑음.', '어제 주가는 큰 폭으로 상승했습니다.']
metadata = [{'category':'test'}, {'category':'test'}]

doc2 = [Document(page_content = input_list2[i], metadata = metadata[i]) for i in range(2)]
ind2 = database.add_documents(doc2)
```

Vector DB

✓ 조회

- 전체 조회

```
database.get()
```

- 인덱스 조회

```
database.get(index)
```

- 조건 조회 : ChromaDB에서는 제공하지 않음 ➔ Dataframe으로 변환 후 사용 가능.

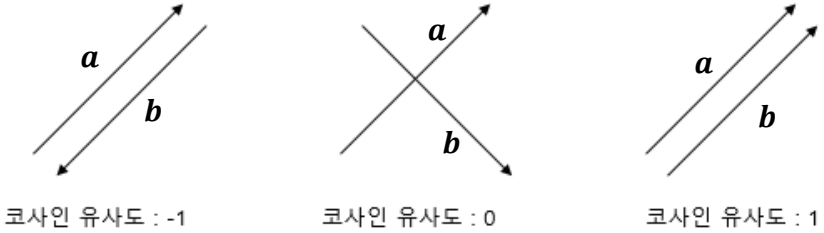
```
data = database.get()
data = pd.DataFrame(data)
data.loc[data['metadatas'] == {'category': 'test'}]
```

유사도

✓ Chroma DB 을 이용하여 검색 시 계산되는 유사도 점수(Similarity Score)

- **Cosine Distance** 이용
  - Cosine Distance $(\boldsymbol{a}, \boldsymbol{b})$ = 1 − Cosine Similarity $(\boldsymbol{a}, \boldsymbol{b})$
  - 0에 가까울 수록 유사도가 높음

- 참고 : 코사인 유사도 Cosine Similarity $(\boldsymbol{a}, \boldsymbol{b}) = \cos\theta = \frac{\boldsymbol{a}}{\|\boldsymbol{a}\|} \cdot \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|}$ $\boldsymbol{a}, \boldsymbol{b}$는 벡터, $\|\boldsymbol{a}\|$: 벡터$\boldsymbol{a}$의 노름($\boldsymbol{norm}$, 길이, 크기)
  - 수학적 의미 : 두 단위 벡터의 내적
  - 자연어 처리에서의 의미 : 유사한 벡터 ➔ 유사한 문맥적 의미



유사도

✓ .similarity_search_with_score()

- query : 질문
- k : 유사도 상위 문서 수

```
# 문서 조회
query = "오늘 낮 기온은?"   # 질문할 문장
k = 3                        # 유사도 상위 k 개 문서 가져오기.

# 데이터베이스에서 유사도가 높은 문서를 가져옴
result = database.similarity_search_with_score(query, k = k)
print(result)

for doc in result:
    print(f"유사도 점수 : {round(doc[1], 5)}, 문서 내용:
{doc[0].page_content}")
```

RAG

✓ RetrievalQA

- RAG용 QA chat 함수
  - llm : 언어 모델
  - retriever : RAG로 연결할 VectorDB
  - return_source_documents=True : 모델이 답변을 생성할 뿐만 아니라 그 답변에 사용된 출처 문서(document)도 함께 반환

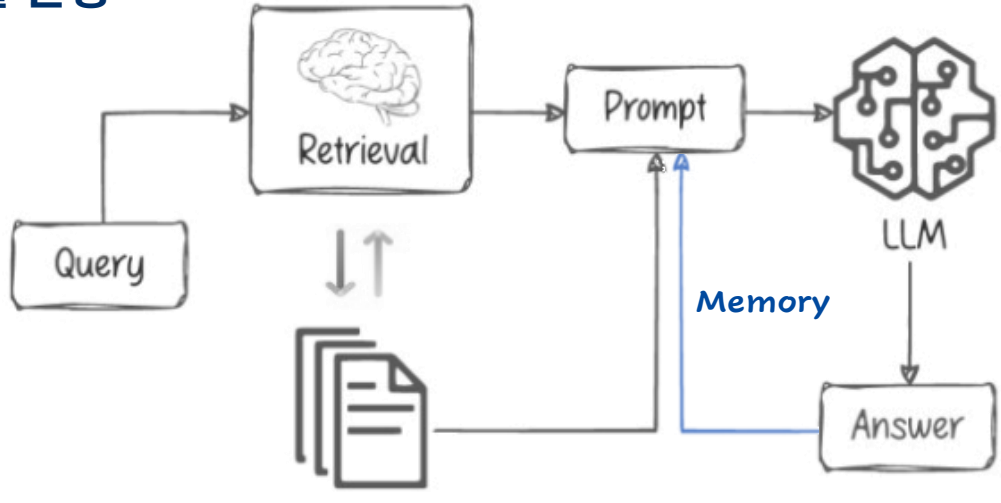
```
chat = ChatOpenAI(model="gpt-3.5-turbo")
retriever = database2.as_retriever()
qa = RetrievalQA.from_llm(llm=chat,  retriever=retriever,  return_source_documents=True )

query = "생성형 AI 도입 시 예상되는 보안 위협은 어떤 것들이 있어?"
result = qa(query)

print(result["result"])
```

5. LangChain③ : RAG - Memory

Memory

✓ 대화의 맥락을 이어가려면...

- 사람 : **이전 대화를 기억**하면서 **현재 대화를 진행**
- 챗봇 :
  - 이전 대화를 기억
    - 이전 질문 답변을 **Memory**에 저장하고
    - 이를 Prompt에 포함
  - 현재 대화 진행
    - 다시 질문



Retrieval-Augmented Generation (RAG)

Memory

✓ ConversationBufferMemory 함수

- 대화를 저장하는 메모리

- .save_context()
  - 딕셔너리 형태로 저장
  - input : HumanMessage
  - output : AIMessage

- .load_memory_variables({})
  - 현재 메모리 내용 전체 조회

```python
from langchain.memory import ConversationBufferMemory

# 메모리 선언하기(초기화)
memory = ConversationBufferMemory(return_messages=True)

# 저장
memory.save_context({"input": "안녕하세요!"},
                    {"output": "안녕하세요! 어떻게 도와드릴까요?"})

memory.save_context({"input": "메일을 써야하는데 도와줘"},
                    {"output": "누구에게 보내는 어떤 메일인가요?"})

# 현재 담겨 있는 메모리 내용 전체 확인
memory.load_memory_variables({})
```

Memory

✓ 채팅 내용 저장

- 질문과 답변을 memory.save_context()로 저장

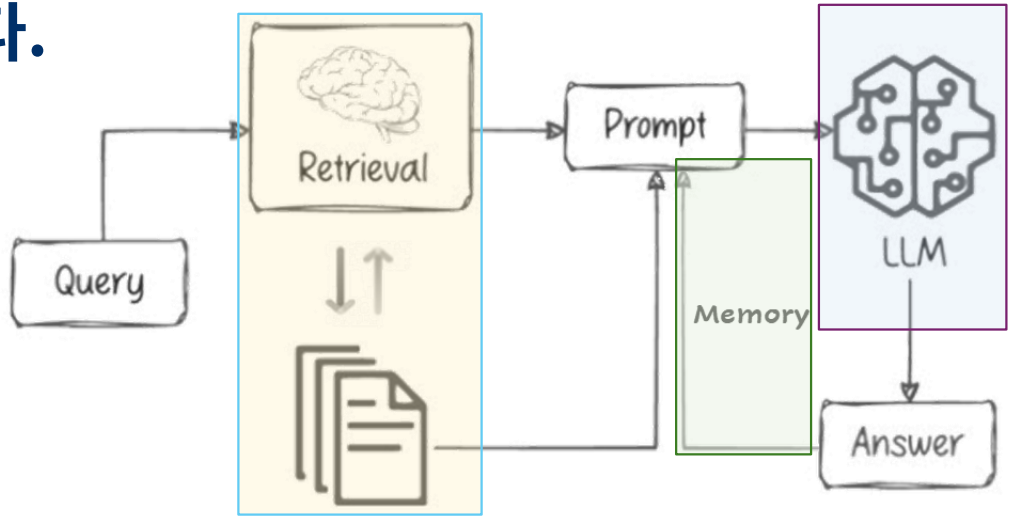
```
qa = RetrievalQA.from_llm(llm=chat,  retriever=retriever,  return_source_documents=True)

query = "생성형 AI 도입시 예상되는 보안 위협은 어떤 것들이 있어?"
result = qa(query)

memory = ConversationBufferMemory(return_messages=True)

memory.save_context({"input": query},
                    {"output": result['result']})

memory.load_memory_variables({})
```

- 맥락이 이어지게 하려면, 프롬프트에 memory 내용이 포함되어야 함.

Chain

✓ **이제 각 모듈을 모두 연결해 봅시다.**

- LLM 모델
- retriever
- memory



Memory

✓ **ConversationalRetrievalChain 함수**

```
qa = ConversationalRetrievalChain.from_llm(llm=chat, retriever=retriever, memory=memory,
                                           return_source_documents=True, output_key="answer")
```

Chain

✓ ConversationalRetrievalChain 함수

- **return_source_documents=True**
  - True로 설정되어 있으면, 답변과 함께 출처 문서도 반환됨
- **Output_key**
  - Output_key 파라미터는 모델의 출력이 저장될 키 지정

```
qa = ConversationalRetrievalChain.from_llm(llm=chat, retriever=retriever, memory=memory,
                                           return_source_documents=True, output_key="answer")
```

Chain

✓ Chain 함수를 위한 memory 설정

- **memory_key :** 메모리에서 대화 기록 저장을 위한 키(이 키를 통해 대화 내역을 저장하고 불러올 수 있음)
- **input_key :** 사용자 입력(질문) 키(이 키를 통해 사용자의 질문이 메모리에 저장됨)
- **Output_key :** 모델의 출력 키(이 키를 통해 모델의 답변이 메모리에 저장됨)
- **return_messages = True** : 메모리에 저장된 대화 내역이 메시지 형식으로 반환 됨

```
memory = ConversationBufferMemory(memory_key="chat_history",
                                  input_key="question",
                                  output_key="answer",
                                  return_messages=True)
```

감사합니다.