RAG 작업량 근거 자료 및 로직 설명

기본 rag 기능 - 방대한 문서를 가지고 검색증강생성을 통해 사용자질문을 시멘틱서치를 하여 문서에서 원하는 문서를 가지고 올수있도록 해줌

셀프쿼리리트리버
기존 리트리버에서 메타데이터로 근거로 필터를 넣을수있음
필터를 통한 1차적인 문서를 필터하고 필터로 줄은 문서안에서 시멘틱서치를 통해 원하는 문서를 보다 더 정확한 문서를 가져올수있음

셀프쿼리리트리버 예시

영화 설명과 메타데이터를 기반으로 유사도 검색이 가능한 벡터 저장소가 구축합니다.

- `Document` 클래스를 사용하여 영화에 대한 간략한 설명과 메타데이터를 포함하는 문서 객체 리스트인 `docs`를 생성합니다.
- `OpenAIEmbeddings`를 사용하여 문서 임베딩을 생성합니다.
- `Chroma.from_documents` 메서드를 사용하여 `docs`와 `OpenAIEmbeddings`로부터 Chroma 벡터 저장소인 `vectorstore`를 생성합니다.

```
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings


docs = [
    Document(
        page_content="A bunch of scientists bring back dinosaurs and mayhem breaks loose",
        metadata={"year": 1993, "rating": 7.7, "genre": "science fiction"},
    ),
    Document(
        page_content="Leo DiCaprio gets lost in a dream within a dream within a dream within a
...",
        metadata={"year": 2010, "director": "Christopher Nolan", "rating": 8.2},
    ),
    Document(
        page_content="A psychologist / detective gets lost in a series of dreams within dreams with
in dreams and Inception reused the idea",
        metadata={"year": 2006, "director": "Satoshi Kon", "rating": 8.6},
    ),
```

SelfQueryRetriever 생성하기

이제 retriever를 인스턴스화할 수 있습니다. 이를 위해서는 문서가 지원하는 **메타데이터 필드** 와 문서 내용에 대한 **간단한 설명을 미리 제공** 해야 합니다.

`AttributeInfo` 클래스를 사용하여 영화 메타데이터 필드에 대한 정보를 정의합니다.

- 장르(`genre`): 문자열 타입, 영화의 장르를 나타내며 ['science fiction', 'comedy', 'drama', 'thriller', 'romance', 'action', 'animated'] 중 하나의 값을 가집니다.
- 연도(`year`): 정수 타입, 영화가 개봉된 연도를 나타냅니다.
- 감독(`director`): 문자열 타입, 영화 감독의 이름을 나타냅니다.
- 평점(`rating`): 실수 타입, 1-10 범위의 영화 평점을 나타냅니다.

```
from langchain.chains.query_constructor.base import AttributeInfo
from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain_openai import ChatOpenAI

metadata_field_info = [
    AttributeInfo(
        name="genre",
        description="The genre of the movie. One of ['science fiction', 'comedy', 'drama', 'thrille
r', 'romance', 'action', 'animated']",
        type="string",
    ),
    AttributeInfo(
        name="year",
        description="The year the movie was released",
        type="integer",
    ),
    AttributeInfo(
        name="director",
        description="The name of the movie director",
        type="string",
    ),
    AttributeInfo(
        name="rating", description="A 1-10 rating for the movie", type="float"
```

`document_content_description` 변수에 영화에 대한 간략한 요약 설명을 할당합니다.

```
# 문서의 내용에 대한 간략한 설명
document_content_description = "Brief summary of a movie"
```

`SelfQueryRetriever.from_llm()` 메서드를 사용하여 `retriever` 객체를 생성합니다.

- `llm`: 언어 모델
- `vectorstore`: 벡터 저장소
- `document_content_description`: 문서 내용 설명
- `metadata_field_info`: 메타데이터 필드 정보

```
# LLM 정의
llm = ChatOpenAI(model="gpt-4-turbo-preview", temperature=0)

# SelfQueryRetriever 생성
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectorstore,
    document_content_description,
    metadata_field_info,
)
```

1. 기존rag에 넣을 데이터document타입에서 메타데이터를 활용 - 카테고리에 대한 메타데이터를 뽑아 기존데이터 가공 - 벡터 디비저장
2. metadata의 info를 입력하여 알게 한다. (name, description,type)형식이야함
3. 벡터디비와 문서의 내용설명 , 메타데이터의 설명, llm을 통해 리트리버를 생성한다.

기존 셀프쿼리리트리버에서 chain을 사용해서 다양한 라이브러리 커스텀을 활용해서 셀프쿼리리트리버 강화 - 보통은 셀프쿼리만 씀

1. 메타데이터의 설명중 유니크한 값들에 대한 명시를 하여 필터의 퀄리티를 높일수있었음
2. 중요한 부분 chain

```
from langchain.chains.query_constructor.base import ( #커스텀 쿼리 프롬프트 작성하기위해 기존

get_query_constructor_prompt,

load_query_constructor_runnable,

)
```

```
doc_contents = "견적에 대한 자세한 설명"

prompt = get_query_constructor_prompt(doc_contents, attribute_info)

print(prompt.format(query="{query}"))
```

커스텀 체인을 구현하여 프롬프트와 필터를 이용하여 현재 견적에 맞게 설정하게 됨

우리의 RAG 로직

랭체인라이브러리설치

```
!pip install langchain openai chromadb tiktoken pypdf unstructured sentence-transformers jq lark langchain-openai
```

랭체인 loader를 이용하여 json파일읽기

```
from langchain.document_loaders import DirectoryLoader
from langchain.document_loaders import JSONLoader

loader = DirectoryLoader('최종데이터', glob="**/*.json", loader_cls=JSONLoader, loader_kwargs = {'jq_schema':'.', 'text_content':False})
docs = loader.load()
```

유니코드데이터에서 분석을 위해 한글인코딩처리

```
def escape_quotes_in_raw_text(json_str):

    start_index = json_str.find('"quote_description": "') + len('"quote_description": "')

    end_index = json_str.find('", "', start_index)

    raw_text_value = json_str[start_index:end_index].replace('"', '\\"')

    fixed_json_string = json_str[:start_index] + raw_text_value + json_str[end_index:]

    return fixed_json_string
```

```
def escape_quotes_in_sentiment_text(json_str):

    search_str = '"quote_feedback": "'
    start_index = 0

    while True:

        start_index = json_str.find(search_str, start_index)
        if start_index == -1:
            break


        start_index += len(search_str)

        end_index = json_str.find('", "', start_index)
        if end_index == -1:
            break


        sentiment_text = json_str[start_index:end_index].replace('"', '\\"')

        json_str = json_str[:start_index] + sentiment_text + json_str[end_index:]


        start_index = end_index + len('", "')

    return json_str
```

파이썬 딕셔리너파일을 df으로 변환 (df로 분석을 위해 변경)

```python
import json

refined_docs = []
for doc in docs:

    python_dict = json.loads(doc.page_content)

    refined_docs.append(python_dict)

# 결과를 확인
for pd in refined_docs:
    print(pd)
```

```python
import pandas as pd

data_df = pd.DataFrame(refined_docs)
data_df
```

숫자데이터를 문자열로 되있어서 숫자int타입으로 컨버터, 벤치마크순위등을 데이터전처리

```python
def convert_price_to_int(price):
    if isinstance(price, str):

        return int(price.replace('원', '').replace(',', ''))
    return price


data_df['total_price'] = data_df['total_price'].apply(convert_price_to_int)

print(data_df[['quote_title', 'total_price']])
```

```
                                          quote_title  total_price
0                                  AMD 라이젠5-4세대 5600       533150
1              인텔 코어i5-13세대 13600KF + 지포스 RTX 4070 SUPER      3344070
2                   AMD 라이젠5-5세대 7500F + 지포스 RTX 4060 Ti      1370160
3                 코어i5-14세대 14600KF + 지포스 RTX 4070 SUPER      2250500
4     로스트아크 검은사막 온라인MMORPG 스팀게임 고사양게임 라이젠 7800X3D + ...      2491800
...                                                 ...          ...
4022  2D작업 캐드 포토샵 일러스트 3D작업 스케치업 라이노 브이레이 i5 13600KF...      2643720
4023       AMD 라이젠7-5세대 7800X3D + 지포스 RTX 4070 Ti SUPER      3630448
4024                                                            0
4025  디아4 옵치 GTA5 롤 풀옵 라이젠 7600 + RTX 4060Ti 화이트감성 F...      1552160
4026               7800X3D + 4070 TI SUPER(NZXT 올블랙 감성)      2810580

[4027 rows x 2 columns]
```

```python
data_df = data_df.dropna(subset=['cpu_score'])
data_df['cpu_score'] = data_df['cpu_score'].astype(int)
unique_values_list = list(data_df['cpu_score'].unique())
print(unique_values_list)
```

```
[504, 194, 355, 181, 225, 98, 312, 257, 107, 595, 63, 247, 366, 209, 56, 60, 381, 84
```

필요없는 카테고리 제거 (title이나 판단하에 필요없는 카테고리 제거)

```python
data_df = data_df[[ 'quote_title', 'date_create', 'cpu_gpu_combinations',
       'quotation_summary', 'quote_person_introduction', 'quote_feedback',
       'computer_estimate_data', 'quote_description', 'parts_price',
       'total_price', 'CPU', 'Motherboard', 'Graphic Card', 'SSD', 'Memory',
       'Power Supply', 'CPU Cooler', 'cpu_score', 'gpu_score',
       'performance_grade']]
data_df
```

```
       quote_title  date_create  cpu_gpu_combinations  quotation_summary  quote_per
```

attributeinfo를 추출하기위한 df.head를 뽑아서 카테고리에 대해 분석하여 설명을 붙이기 (메타데이터 생성) (name,description,type순으로)

```
from langchain_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4", openai_api_key=OPENAI_KEY)
res = model.predict(
    "Below is a table with information about Description of computer parts."
    "Return a JSON list with an entry for each column. Each entry should have "
    '{"name": "column name", "description": "column description", "type": "column data type"}'
    f"\n\n{data_df.head()}\n\nJSON:\n"
)
```

attributeinfo추가 (프롬프트에 추가하여 메타데이터의 정확성을 명시하게 됨)

```
import json

attribute_info = json.loads(res)
attribute_info
```

```
[{'name': 'quote_title',
  'description': 'Title of the quote for the computer configuration',
  'type': 'string'},
 {'name': 'date_create',
  'description': 'Date the quote was created',
  'type': 'date'},
 {'name': 'cpu_gpu_combinations',
  'description': 'Combinations of CPU and GPU used in the computer',
  'type': 'string'},
 {'name': 'quotation_summary',
  'description': 'Brief summary of the quote',
  'type': 'string'},
 {'name': 'quote_person_introduction',
  'description': 'Introduction of the person who made the quote',
  'type': 'string'},
 {'name': 'quote_feedback',
  'description': 'Feedback received on the quote',
  'type': 'string'},
 {'name': 'computer_estimate_data',
  'description': 'Data on the estimated cost of the computer configuration',
  'type': 'string'},
 {'name': 'quote_description',
  'description': 'Detailed description of the quote',
  'type': 'string'},
 {'name': 'parts_price',
  'description': 'Price of the individual parts used in the computer configuration',
  'type': 'dictionary'},
 {'name': 'total_price',
  'description': 'Total price of the computer configuration',
  'type': 'integer'},
 {'name': 'CPU',
  'description': 'Information on the CPU used in the computer configuration',
  'type': 'string'},
 {'name': 'Motherboard',
  'description': 'Information on the motherboard used in the computer configuration',
  'type': 'string'},
```

데이터 유니크한값은 attributeinfo에 명시하면 더 정확하게 알려줄수있음(분석한 유니크한 값들을 메타데이터 설명에 추가하여 더 확실하게 명시하게 됨)

```python
attribute_info[-3][
    "description"
] += f". Valid values are {sorted(data_df['cpu_score'].value_counts().index.tolist())}"
attribute_info[-2][
    "description"
] += f". Valid values are {sorted(data_df['gpu_score'].value_counts().index.tolist())}"
attribute_info[-1][
    "description"
] += f". Valid values are {sorted(data_df['performance_grade'].value_counts().index.tolist())}"
```

```
 {'name': 'cpu_score',
  'description': 'Score of the CPU used in the computer configuration. Valid values are [0, 1, 11, 13, 14, 23, 24, 56, 60, 63, 67, 98, 107, 111, 136, 181, 194, 209, 225, 247, 257, 312,
332, 335, 355, 359, 361, 366, 399, 504, 595, 844]',
  'type': 'integer'},
 {'name': 'gpu_score',
  'description': 'Score of the GPU used in the computer configuration. Valid values are [0, 1, 4, 6, 11, 14, 25, 28, 31, 32, 37, 43, 47, 53, 67, 72, 80, 81, 88, 106, 132, 152, 153, 170,
183, 184, 196, 207]',
  'type': 'integer'},
 {'name': 'performance_grade',
  'description': "Performance grade of the computer configuration. Valid values are ['고성능', '저성능', '중성능']",
  'type': 'string'}]
```

chain을 생성해서 커스텀 쿼리 프롬프트를 구현

자세하게 이야기

- attribute_info와 스키마를 설정하여(메타데이터카테고리에 대한 설명과 스키마 규칙을 설정 - 랭체인 공식문서를 참고하여 가능한 스키마를 추출한뒤에 그 스키마로 어떻게 메타데이터로 거를수있는지 생각 후 스키마 규칙생성)
- 랭체인 공식문서 참고

langchain_core.structured_query.Comparator

class langchain_core.structured_query.**Comparator**(*value*) [source]

Enumerator of the comparison operators.

EQ = *'eq'*

NE = *'ne'*

GT = *'gt'*

GTE = *'gte'*

LT = *'lt'*

LTE = *'lte'*

CONTAIN = *'contain'*

LIKE = *'like'*

IN = *'in'*

NIN = *'nin'*

Examples using Comparator

- constructing-filters.md

langchain_core.structured_query.Operator

class langchain_core.structured_query.**Operator**(*value*) [source]

Enumerator of the operations.

AND = *'and'*

OR = *'or'*

NOT = *'not'*

Examples using Operator

- constructing-filters.md

- 이러한 허용 스키마를 찾아보고 여러문서를 참고하여 스키마 규칙생성
- 어떤 랭체인을 이용해서 커스텀했는지

```
from langchain.chains.query_constructor.base import (
    get_query_constructor_prompt,
    load_query_constructor_runnable,
)
```

```
prompt = get_query_constructor_prompt(doc_contents, attribute_info)
```

스키마규칙 , 메타데이터카테고리의설명
결국 스키마 규칙

```
Your goal is to structure the user's query to match the request schema provided below.

<< Structured Request Schema >>
When responding use a markdown code snippet with a JSON object formatted in the following schema:

```json
{
    "query": string \ text string to compare to document contents
    "filter": string \ logical condition statement for filtering documents
}
```

The query string should contain only text that is expected to match the contents of documents. Any conditions in the filter should not be mentioned in the query as well.

A logical condition statement is composed of one or more comparison and logical operation statements.

A comparison statement takes the form: `comp(attr, val)`:
- `comp` (eq | ne | gt | gte | lt | lte | contain | like | in | nin): comparator
- `attr` (string): name of attribute to apply the comparison to
- `val` (string): is the comparison value

A logical operation statement takes the form `op(statement1, statement2, ...)`:
- `op` (and | or | not): logical operator
- `statement1`, `statement2`, ... (comparison statements or logical operation statements): one or more statements to apply the operation to

Make sure that you only use the comparators and logical operators listed above and no others.
Make sure that filters only refer to attributes that exist in the data source.
Make sure that filters only use the attributed names with its function names if there are functions applied on them.
Make sure that filters only use format `YYYY-MM-DD` when handling date data typed values.
Make sure that filters take into account the descriptions of attributes and only make comparisons that are feasible given the type of data being stored.
```

- get_query_constructor_prompt로 프롬프트 작성(스키마,attribute_info) -> load_query_constructor_runnable 위 커스텀 프롬프트로 실제로 필터와 쿼리를 생성하게끔 도와줌 ,커스텀 프롬프트를 참고하여 LLM이 필터생성해줌 ->

```
(
    "200만원 이하의 게이밍 PC를 추천해줘.",
    {
        "query": "200만원 이하 게이밍 PC 견적",
        "filter": "and(gte('total_price', 1800000), lte('total_price', 2000000))"
    }
),
(
    "100만원대의 사무용 컴퓨터 견적을 보여줘.",
    {
        "query": "사무용",
        "filter": "and(gte('total_price', 900000), lte('total_price', 1100000)"
    }
),
 (
    "300만원 이내의 고성능 그래픽 작업용 PC를 추천해줘.",
    {
        "query": "300만원이내, 그래픽 작업용",
       "filter": "and(gte('total_price', 2800000), lte('total_price', 3000000),eq('performance_grade','고성능'))"
 }
),
```

- filter의 할루시네이션(query내용이 영어로 갑자기 바뀌거나 filter에서 300만원 이하일경우 100만원도 추천해주는거를 방지하기위해 더 명시적으로 프롬프트에 추가함
- 할루시네이션 예시

```
chain.invoke({"query": "고성능 견적 추천해줘"})
```

```
StructuredQuery(query='high performance', filter=None, limit=None)
```

이런식으로 대표적인 명시를 통해 필터의 유효성을 올리게 됨 -> 다시 load_query_constructor_runnable을 이용하여 필터,쿼리예시까지 추가하여 체인생성 -> invoke를 통한 필터테스트

```
chain.invoke({"query": "170만원 견적 추천을 해줘"})
```

```
StructuredQuery(query='170만원 견적', filter=Operation(operator=<Operator.AND: 'and'>, arguments=[Comparison(comparator=<Comparator.GTE: 'gte'>, attribute='total_price', value=1600000),
Comparison(comparator=<Comparator.LTE: 'lte'>, attribute='total_price', value=1700000)]), limit=None)
```

->

필터의 정확도 상승 및 기존 가격에 대한 명시가 없어 170만원이하면 50만원도 보여주는 오류?가 떠서 리미트를 잡아 2,30만원에 대한 리미트를 설정하였기 때문에 저런 필터가 만들어짐 ->

허깅페이스를 통한 gpu 임베딩모델설정

```
from langchain_community.vectorstores import Chroma
from langchain.embeddings import HuggingFaceEmbeddings

model_name = "jhgan/ko-sroberta-multitask" # (KorNLU 데이터셋에 학습시킨 한국어 임베딩 모델)
model_kwargs = {'device': 'mpu'}
encode_kwargs = {'normalize_embeddings': False}
embeddings = HuggingFaceEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs
)
```

현재 data_df에 있던 것을 document타입으로 설정하여 랭체인 인식이 되게끔 바꾸게 됨

```
import json
import pandas as pd
from langchain_core.documents import Document

# 가정: data_df는 이미 적절히 정의되어 있고, 사용할 준비가 되어 있음
docs = []
for _, data in data_df.fillna("").iterrows():
    data_dict = data.to_dict()
    print(data_dict)

    # 특정 키를 제외한 나머지 데이터를 메타데이터로 저장
    keys_to_exclude = {"parts_price"}
    aspect_exclude_data_dict = {k: v for k, v in data_dict.items() if k not in keys_to_exclude}

    # Document 객체 생성
    doc = Document(
        page_content=json.dumps(data_dict, indent=2, ensure_ascii=False),
        metadata=aspect_exclude_data_dict
    )
    docs.append(doc)
```

docs에는 document타입으로 변환된 견적데이터와 임베딩모델을 설정해서 chroma라이브러리를 이용하여 벡터디비를 임베딩하여 벡터디비만들게됨

```
vecstore = Chroma.from_documents(docs, embeddings)
```

selfqueryretriever랭체인 라이브러리를 가져와 , 커스텀쿼리(체인)와 벡터디비,k값설정하여 리트리버설정

```
from langchain.retrievers import SelfQueryRetriever

retriever = SelfQueryRetriever(
    query_constructor=chain, vectorstore=vecstore, verbose=True,k=4
)
```

리트리버를 통해 견적추천받기

```python
results = retriever.get_relevant_documents(
    "100만원정도의 저성능 컴퓨터 견적을 추천해줘"
)

for res in results:
    result_dict = json.loads(res.page_content)
    print(result_dict['quote_title'])
print('--------------------------------------------------------------------------------')

for res in results:
    print(res.page_content)
    print("\n" + "-" * 20 + "\n")
```

```
      "수량": "1"
    }
  },
  "total_price": 1009000,
  "CPU": "인텔 코어i5-12세대 12400F (엘더레이크)(벌크)",
  "Motherboard": "ASRock B760M PG LIGHTNING/D4 에즈윈",
  "Graphic Card": "ZOTAC GAMING 지포스 RTX 4060 TWIN Edge OC D6 8GB",
  "SSD": "장우컴퍼니 PM9A1 M.2 NVMe 병행수입 (1TB) 삼성전자",
  "Memory": "ESSENCORE KLEVV DDR4-3200 CL22 (16GB) x 2 개",
  "Power Supply": "FSP HYPER K PRO 600W 80PLUS Standard 230V EU",
  "CPU Cooler": "3RSYS TEAMMOST TM40",
  "cpu_score": 595,
  "gpu_score": 32,
  "performance_grade": "저성능"
}

--------------------
```

```
인텔 FHD 가성비 게이밍 PC
인텔 i5, RTX 3060 배그, 디아블로 가성비 PC 견적
2024년 1월 월간 견적 – 중급 FHD 게이밍 (인텔+RTX 4060)
2024년 03월 월간 견적 – 상급 FHD 게이밍 (인텔 + RX 7600)
```

견적데이터정제기간: 1주

구현기간 : 약 3,4주

코드줄 :

```
        for res in results:
            result_dict = json.loads(res.page_content)
            print("Found result:", result_dict['quote_title'])
    else:
        print("No results found.")

# 쿼리 실행
test_query = {
    "query": "170만원짜리 컴퓨터 견적",
}
debug_search(test_query)

# 쿼리 생성
query = "롤할 정도의 컴퓨터 견적을 추천해줘"
structured_query = {
    "query": query,
    "filter": 'contain("quote_description", "롤")'
}

# 검색 실행
results = retriever.invoke(structured_query)

if results:
    for res in results:
        # 결과가 튜플로 반환되고 첫 번째 요소가 문서 객체인 경우
        result_dict = json.loads(res[0].page_content)  # 인덱스 0을 사용하여 접근
        print(result_dict['quote_title'])
    print('--------------------------------------------------------------------------------')

    for res in results:
        print(res[0].page_content)  # 결과 출력을 위해 첫 번째 요소 접근
        print("\n" + "-" * 20 + "\n")
else:
    print("No results found.")
```

약 1400줄

웹개발 플라스크 + react
구현기간 : 약 1주
코드줄: 900줄 + 1200줄

aws 랭체인코드 ec2에 올려서 api로 쓰는거 (안정화까지)
구현기간: 2주

LLM파인튜닝 및 데이터정제기간
구현기간: 3주

시간단축
파인튜닝한 gpt api호출을 처음 견적데이터를 짤라서 넣어서 약 9번 for문으로 돌려서 시간복잡도가 높다고 판단
-> 멀티프로세스으로 api를 비동기로 호출하여 약 평균 40초에서 20초로 줄임