HTP 검사 해석 RAG 최종 평가

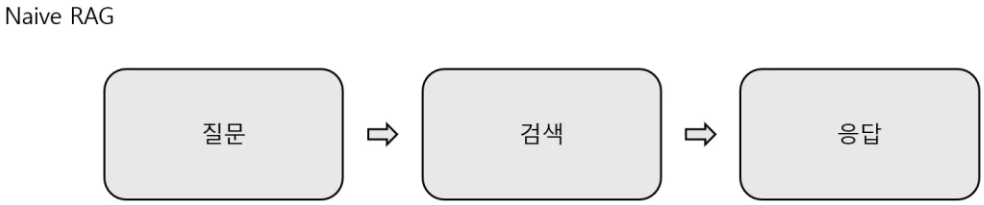
박진호

목차

1. RAG pipeline 개선 사항(LangGraph 도입)
2. Naive RAG vs Graph RAG 비교
3. 최종 평가 결과

1. RAG pipeline 개선 사항(LangGraph 도입)

기존 Naive RAG 의 한계점



1. 단방향 체인

Langchain 을 활용하여 RAG 를 구축하면, flow 가 단방향으로 구성된다. 단방향의 체인의 문제점은 이전 분기로 다시 돌아갈 수 없다는 것이다. 예를 들어서 전혀 다른 질문이 들어와도 파이프라인에서 검색과 생성이 진행된다.

2. 분기 별 진행 여부 판단 불가

질문이 적절한지, 검색은 적절한지 판단하기가 모호하다. 단순히 프롬프트에 금지 사항을 작성하는 것으로는 유저가 만족할만한 결과가 나오지 않을 수 있다. 그리고 항상 생성단계까지 거치기 때문에, context 가 그대로 길어져 cost 와 시간의 소모가 일정하게 사용된다.

3. 사용자 경험이 떨어짐

사용자가 의도에 맞지 않는 질문을 입력하거나 했을 때, 또 잘못된 내용이 검색되었을 때 잘못된 정보가 생성될 가능성이 매우 높아진다.

Graph RAG 도입

노드(분기)와 엣지를 통해 그래프 형태로 RAG 파이프라인을 구축한다. 각 노드에서 로직을 수행하고 분기를 어디로 향하게 할 지 정하여 좀 더 유연하고, 정확도가 높은 RAG 를 구축할 수 있다. 사용자의 입력과 검색을 검사하여 수정을 유도하고 의도에 맞는 응답을 생성해낼 수 있다. 좀 더 agentic 한 pipeline 을 구축하는 방법이다.

사용한 framework: LangGraph

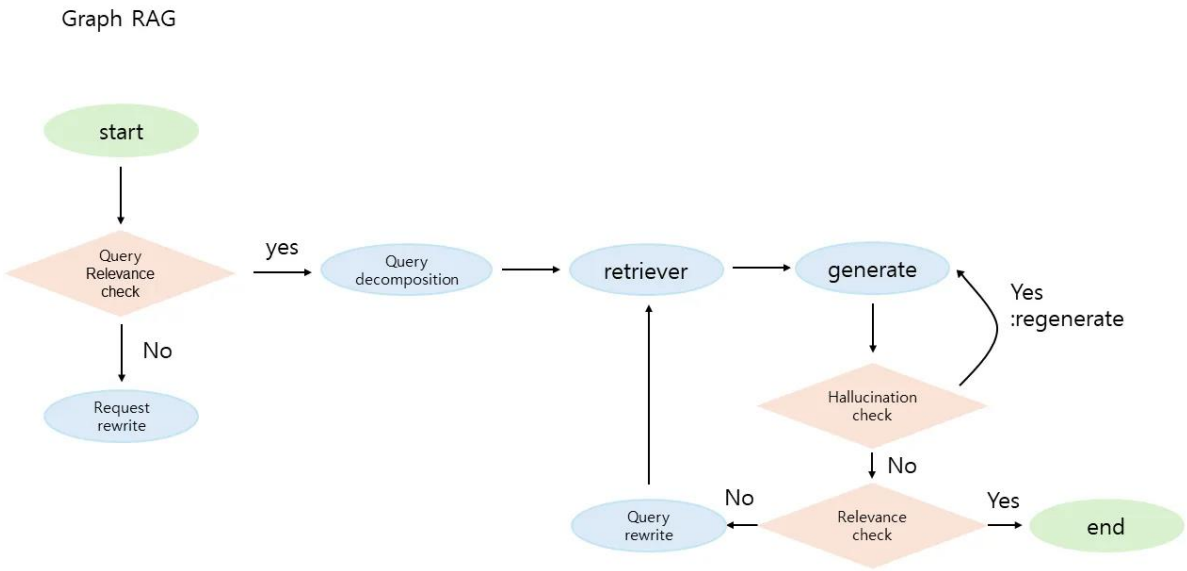
LangGraph 는 node 와 edge 를 관리하고 workflow 를 유연하게 연결하기 쉽게 도와주는 프레임워크이다. 도입 이유는 다음과 같다.

1. 높은 유저경험
2. 높은 정확도

HTP 심리검사는 특히 정확한 결과가 중요하다. 우리의 가장 중요한 목표는 정확도이고, 정확도가 높아지면 유저 경험이 높아진다고 판단하였다.

고려 사항

Graph RAG 를 도입하면, 분기(Node)를 거치면서 진행하기 때문에 지연시간과 비용이 증가하게 된다. 요구사항에 맞는 의사결정을 위해 Navie RAG 와 Graph RAG 의 지연시간과 비용을 비교하여서 합리적은 의사 결정을 할 필요성이 있다.



Graph 설계 초안

2. Naive RAG vs Graph RAG 비교

비교 항목

- 지연 시간
- 비용

0. 공통사항

생성 모델(GPT-4o-mini), 프롬프트, 검색기(ensemble retriever)

공통질문

inputs = {

"original_question": "집에 창문은 2 개 존재하고 크기는 적절함. 문은 집의 크기에 비해 작으며 문과 바깥이 길로 이어져 있지 않음."

}

interupt_inputs = {

"original_question": "오늘의 날씨는? 대한민국의 수도는? 최근 야구 경기 결과는?"}

1. Latency

1-1. Naive RAG

Flow: 질문 분해 → 질문 검색 → 응답 생성

정상 질문

```
Execution time for generate_node: 18.33 seconds
--- Individual Node Execution Times ---
- decompose_query_node: 2.0267 seconds
- retrieve_node: 1.4572 seconds
- generate_node: 18.3270 seconds

----------------------------------------
Total execution time (sum of nodes): 21.8108 seconds
```

잘못된 질문

```
Execution time for generate_node: 19.42 seconds
--- Individual Node Execution Times ---
- decompose_query_node: 2.0044 seconds
- retrieve_node: 1.1217 seconds
- generate_node: 19.4205 seconds

----------------------------------------
Total execution time (sum of nodes): 22.5465 seconds
```

3 회 응답 평균 약 22 초

1-2. Graph RAG

Flow: 질문 관련성 check → 질문 분해 → 질문 검색 → 응답 생성 → 환각 검사

정상 질문

```
--- Individual Node Execution Times ---
- relevance_check_node: 0.7708 seconds
- decompose_query_node: 1.9607 seconds
- retrieve_node: 3.6678 seconds
- generate_node: 16.6939 seconds
- hallucination_check_node: 0.6193 seconds

----------------------------------------
Total execution time (sum of nodes): 23.7125 seconds
```

잘못된 질문

```
--- Individual Node Execution Times ---
- relevance_check_node: 0.7283 seconds

----------------------------------------
Total execution time (sum of nodes): 0.7283 seconds
```

정상질문 3 회 응답 평균 23 초

잘못된 질문 3 회 응답 평균 0.7 초

2. Cost

2-1. Naive RAG

정상, 비정상 질문 동일

Before

After

평균 10credit 사용

2-2. Graph RAG

정상 질문

Before

After

평균 16credit 사용

- 비정상 질문

Relevance Check 에 1credit 사용

3. 결론

3-1 3 회 평균 응답 생성 시간

Naive RAG: 약 22 초

Graph RAG: 약 23 초

특이사항: relevance, hallucination check 에서 큰 시간이 소모되지 않았다. 평균 각각 약 (0.7 초)

3-2 3 회 평균 비용

Naive RAG: 10credit

Graph RAG: 16credit

특이사항: Naive RAG 에서는 잘못된 질문이 입력되어도 항상 응답을 생성하기에 항상 10credit 을 소모해야 한다. 하지만 Graph RAG 는 잘못된 응답이 입력되었을 때 분기가 종료되기 때문에 1credit 만 소모하는 결과를 확인하였다.

추가되는 분기점 마다 gpt-4o-mini 기준 평균적으로 2 credit 을 소모함을 볼 수 있었다.

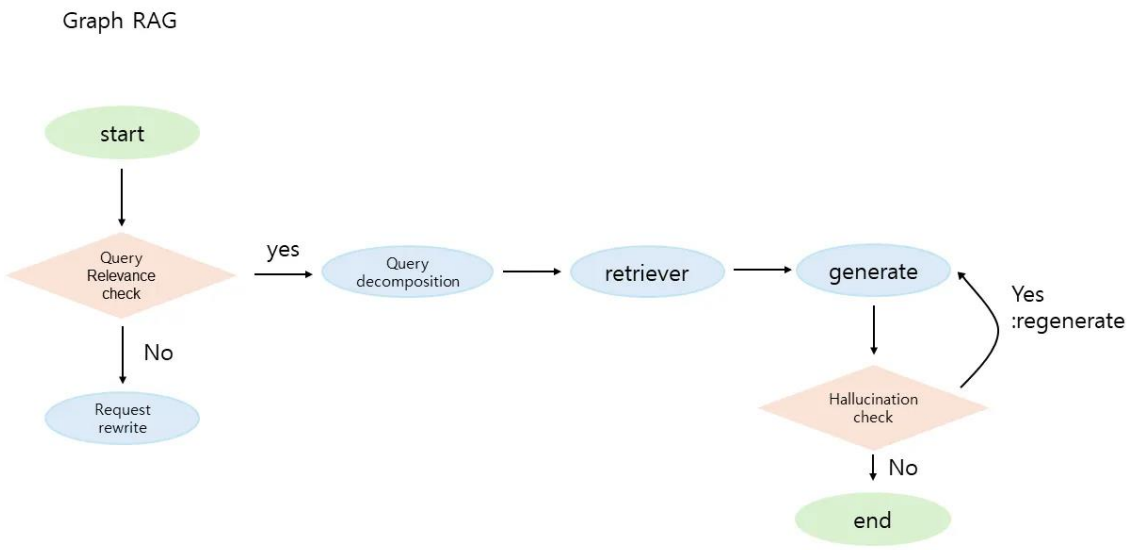
정리

yes or no 로 응답을 하는 분기는 생성에 있어서 시간이 크게 소모되지 않아 Latency 를 크게 고려할 필요는 없었다. 시간적으로 보면 Graph 로 RAG 를 구축하는 것이 항상 이점이 있었다. 특히 잘못된 입력이나 어떤 조건에 의해서 End 로 갔을 때 큰 시간적 낭비를 아낄 수 있었다.

반대로 Cost 는 추가되는 분기마다 선형적으로 2~3 credit 이 증가하였다. 좀 더 Agentic 하게 구현하기 위해서는 많은 코스트의 비용을 감당해야했다.
요구사항을 정확하게 이해하고 꼭 필요한 분기만 추가하도록 고려해야 함을 알게 되었다. 그래도 잘못된 입력이 입력되거나 했을 때, Graph 가 종료되기 때문에 비용을 크게 아낄 수 있었다.

시간보다 가장 비용 효율적인 분기를 추가하여 Cost 낭비를 최소화 하는게 목표

최종 그래프 설계



Hallucination 검사에서 relevance 검사와 겹치는 부분이 있어 과감하게 정리하여 가지를 치고, 이미 recall 이 0.9 이상 나오고 있기 때문에 재 검색 단계는 생략하기로 하였다. Credit 을 약 10% 절약할 수 있는 효과를 기대할 수 있었다.

3. 최종 평가 결과

Graph RAG 를 RAGAs 라이브러리를 통해 평가해 보았다. 실제 쿼리 분해까지 고려한 Ground Truths 를 논문에 작성된 임상 정보와 나뉘어진 청크를 활용하여 Q&A set 20 개를 제작하였다. 사용된 LLM 은 Gemini-2.5-pro 를 사용하였다. 최종 Ground Truths 를 사용하여 이루어진 평가는 다음과 같으며, 초기 목표치인 recall 0.9 를 달성한 것을 확인할 수 있었다.

```
{
  'faithfulness': 0.8597,
  'context_precision': 0.9480,
  'context_recall': 0.9323,
  'semantic_similarity': 0.8338,
}
```

프로젝트를 진행하면서 아쉬웠던 점

프로젝트 요구사항에 맞는 최종응답이 정말로 적절한지 평가할 수 없었다. 근거 있는 논문을 기반으로 RAG 를 구성했다고 하여도, 결국 최종 응답이 얼마나 적절한지는 전문가의 의견과 견해가 필요하기에 적절한 평가 방법을 고안해내야 함을 느꼈다. LLM as a judge 를 통해 최종응답 비교를 하여 더 나은 선택을 하려고 하였지만, 설계의 실패로 편향이 존재했고 제대로 된 평가를 진행하지 못하여 retriever 평가에 좀 더 집중하였다.

프롬프트에 좀 더 집중해야 함을 느꼈다. LLM 의 모델 성능이 좋아지면서 맥락 이해도가 많이 상승하였는데, few shot 을 통해 작성된 프롬프트를 맹신하였다가 이 예제가 편향을 일으킬 수 있다는 사실을 알게 되었다. 예를 들어 추가적인 그림 특징을 추출하지 못했을 때, 예시에 작성된 특징을 참고해버리는 문제가 발생하였다. 프롬프트작성을 좀 더 신중하게 해야 한다고 느꼈다.

정확도를 위해 수작업으로 문서화와 파싱을 하였는데, 너무 많은 시간을 쏟았다. 이 부분에서 어느 정도 자동화된 파이프라인을 찾아야 했다고 생각한다. RAG 문서화에 너무 집중하여 시간을 많이 소모하였고, 이는 이후 최종응답 평가와 프롬프트최적화에 신경을 쓰기 어려운 문제와 직결되었다.

정리하자면,

1. RAG 문서 청킹 자동화 파이프라인을 고려하여야 한다.
2. LLM 의 최종 응답에 대한 평가방법을 고안해야 한다.
3. 프롬프트를 좀 더 최적화하여 응답과 cost 를 최적화해야 한다.

다음 부터는 기타 상용 tool 을 활용하여 좀 더 생산성 있는 방법을 고안하고 그러면서도 정확도를 높이는 방법을 찾아 프로젝트를 진행하려고 한다.