基于 RAG 的医疗问答系统实验报告

目录

---

1. 项目概述

1.1 研究背景

在医疗健康领域，患者常常面临以下问题：

- 专业医学术语难以理解
- 网络信息良莠不齐，难以甄别
- 缺乏及时、可靠的医疗咨询渠道

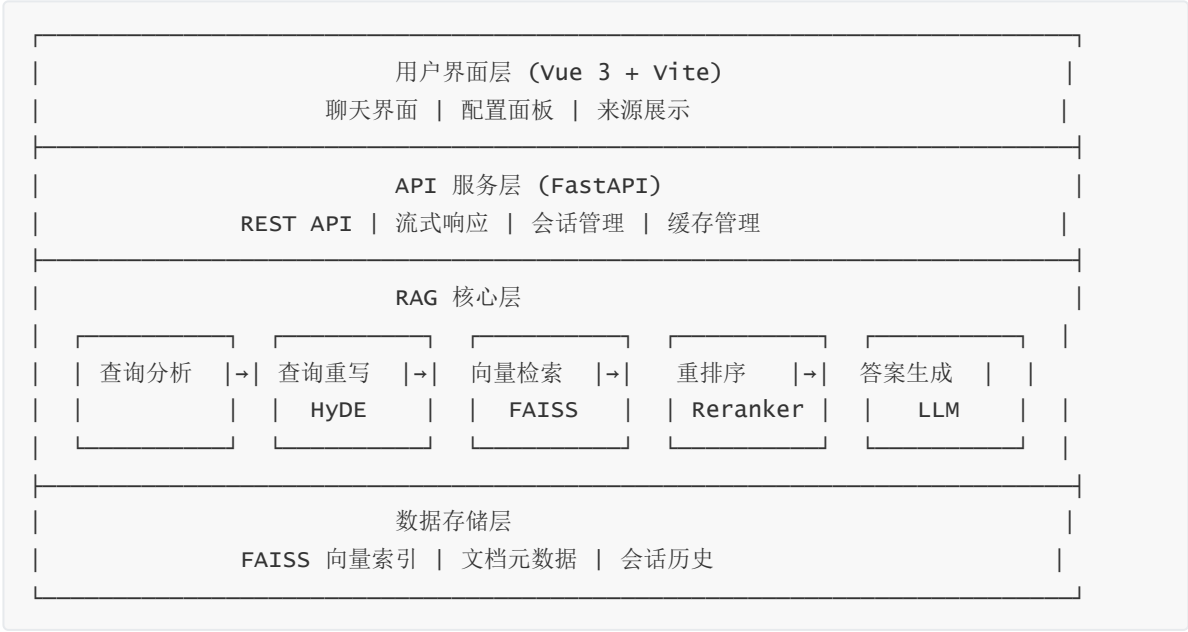
传统搜索引擎基于关键词匹配，难以理解用户的真实意图；而纯粹的大语言模型（LLM）虽然具有强大的语言理解和生成能力，但存在"幻觉"问题，可能生成不准确甚至有害的医疗建议。

1.2 研究目标

本项目旨在构建一个**可靠、准确、可解释**的医疗问答系统，主要目标包括：

1. **提高回答准确性**：通过检索增强生成（RAG）技术，让 LLM 基于真实医学知识库生成回答
2. **增强可解释性**：提供参考来源，让用户了解答案的依据
3. **优化检索效果**：通过查询重写、HyDE、重排序等技术提升检索精度
4. **支持实际应用**：提供完整的前后端系统，支持多轮对话和流式输出

1.3 系统架构



1.4 技术栈

层级	技术选型
前端	Vue 3 + Pinia + Vite + Axios
后端	FastAPI + Uvicorn + Pydantic
向量检索	FAISS + Sentence-Transformers
Embedding	BAAI/bge-base-zh-v1.5
重排序	BAAI/bge-reranker-base
LLM	DeepSeek

---

2. 数据来源与处理

2.1 数据来源

本项目使用的医疗问答数据主要来源于：

数据集	来源	规模	说明
cMedQA2	中文医疗问答社区	10万+ QA对	真实用户问答数据

2.2 数据预处理

2.2.1 数据清洗

```
def clean_text(text):
    # 1. 去除 HTML 标签
    text = re.sub(r'<[^>]+>', '', text)
    # 2. 统一标点符号
    text = text.replace('，', ',').replace('。', '.')
    # 3. 去除多余空白
    text = re.sub(r'\s+', ' ', text).strip()
    # 4. 过滤过短内容
    if len(text) < 10:
        return None
    return text
```

2.2.2 文本分块 (Chunking)

采用递归字符分割策略，保证语义完整性：

```
class TextChunker:
    def __init__(
        self,
        chunk_size: int = 512,      # 目标块大小
        chunk_overlap: int = 50,     # 重叠大小
        separators: list = None      # 分隔符优先级
    ):
        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap
```

```
        self.separators = separators or [
            "\n\n",  # 段落
            "\n",    # 换行
            "。",    # 句号
            "；",    # 分号
            "，",    # 逗号
            " "      # 空格
        ]
```

分块策略说明：

- 优先在段落边界分割，保持语义完整
- 使用重叠窗口避免上下文丢失
- 记录分块元数据（来源文档、位置等）

2.2.3 向量化

使用 `BAAI/bge-base-zh-v1.5` 模型进行文本向量化：

```
class Embedder:
    def __init__(self, model_name="BAAI/bge-base-zh-v1.5"):
        self.model = SentenceTransformer(model_name)
        self.dim = self.model.get_sentence_embedding_dimension()

    def embed(self, texts: List[str]) -> np.ndarray:
        embeddings = self.model.encode(
            texts,
            normalize_embeddings=True,  # L2 归一化
            show_progress_bar=True
        )
        return embeddings.astype(np.float32)
```

模型选择理由：

- BGE 系列在中文语义理解任务上表现优秀
- 768 维向量平衡了表达能力和计算效率
- L2 归一化后可使用内积计算余弦相似度

2.3 数据统计

指标	数值
原始 QA 对数量	（待填写）
处理后文档块数量	（待填写）
平均块长度	（待填写）字符
向量索引大小	（待填写）MB

3. 方法

3.1 基础 RAG 流程

标准 RAG（Retrieval-Augmented Generation）流程：

```
用户查询 → 查询编码 → 向量检索 → 上下文构建 → LLM 生成 → 返回答案
    ↓          ↓          ↓          ↓          ↓
  "头痛"    [0.1,0.2,..] → Top-K文档 → Prompt拼接 → "头痛可能..."
```

核心代码实现：

```python
def query(self, question: str, top_k: int = 5):
    # 1. 检索相关文档
    results = self.retrieve(question, top_k=top_k)

    # 2. 构建上下文
    context = "\n\n".join([
        f"[{i+1}] {r.chunk.content}"
        for i, r in enumerate(results)
    ])

    # 3. 构建提示词
    prompt = f"""基于以下参考资料回答用户问题。

参考资料：
{context}

用户问题：{question}

请给出准确、专业的回答："""

    # 4. 调用 LLM 生成答案
    answer = self.llm.generate(prompt)

    return answer, results
```

3.2 查询重写 (Query Rewriting)

用户查询往往口语化、模糊，需要优化后再进行检索。

3.2.1 单查询重写 (Single)

将用户口语化查询转换为专业检索词：

```
原始查询: "头疼怎么办"
重写后:   "头痛的原因、症状及治疗方法"
```

Prompt 模板：

```
你是一个医学搜索查询优化专家。请将用户的口语化查询改写为更专业的医学检索查询。

用户查询: {query}

要求:
1. 使用专业医学术语
2. 扩展可能的相关概念
3. 保持查询简洁（15-30字）

优化后的查询:
```

3.2.2 多查询扩展 (Multi)

生成多个查询变体，提高召回率：

```
原始查询: "糖尿病怎么治"
扩展查询:
  1. "糖尿病的治疗方法和药物"
  2. "2型糖尿病的控制方案"
  3. "糖尿病患者的饮食管理"
```

多查询检索后进行结果合并和去重：

```python
def retrieve_multi_query(self, queries: List[str], top_k: int):
    all_results = {}
    for query in queries:
        results = self.retrieve(query, top_k=top_k)
        for r in results:
            key = r.chunk.id
            if key not in all_results or r.score > all_results[key].score:
                all_results[key] = r

    # 按分数排序返回 Top-K
    return sorted(all_results.values(), key=lambda x: x.score, reverse=True)
[:top_k]
```

3.2.3 上下文感知重写 (Context)

在多轮对话中，结合历史上下文补全指代和省略：

```
历史对话:
  用户: 高血压有什么症状?
  助手: 高血压早期通常没有明显症状...

当前查询: "那怎么治疗呢？"
重写后:   "高血压的治疗方法有哪些"
```

3.3 HyDE (Hypothetical Document Embeddings)

HyDE 是一种创新的检索方法，通过生成假设性文档来弥合查询与文档之间的语义鸿沟。

3.3.1 核心思想

```
传统方法: Query → Embedding → 检索
HyDE:     Query → LLM生成假设文档 → Embedding → 检索
```

原理：用户查询通常简短，而知识库文档详细。直接用查询向量检索可能因词汇不匹配而失败。HyDE 先用 LLM 生成一个"理想的回答文档"，再用该文档的向量进行检索。

3.3.2 实现

```python
def generate_hyde_document(self, query: str) -> str:
    prompt = f"""请针对以下医学问题，撰写一段专业的回答文档。
这段文档应该是一个理想的、全面的医学解答。

问题: {query}

请直接输出文档内容（100-200字）:"""

    return self.llm.generate(prompt)

def retrieve_with_hyde(self, query: str, top_k: int):
    # 1. 生成假设文档
    hyde_doc = self.generate_hyde_document(query)

    # 2. 用假设文档向量检索
    hyde_embedding = self.embedder.embed([hyde_doc])[0]
    results = self.vector_store.search(hyde_embedding, top_k)

    return results
```

3.3.3 示例

```
原始查询: "头痛吃什么药"

HyDE 生成的假设文档:
"头痛是一种常见症状，治疗需要根据病因选择合适的药物。对于紧张性头痛，
可以使用非甾体抗炎药如布洛芬、对乙酰氨基酚等。偏头痛可使用曲普坦类
药物。如果头痛伴有发热，应考虑解热镇痛药。长期反复头痛建议就医检查，
排除器质性病变。用药前应阅读说明书，注意禁忌症..."

→ 使用该文档向量进行检索，能匹配到更多相关的药物治疗文档
```

3.4 重排序 (Reranking)

向量检索基于语义相似度，但可能引入噪声。使用 Cross-Encoder 重排序模型对初检结果进行精排。

3.4.1 Bi-Encoder vs Cross-Encoder

方法	原理	优点	缺点
Bi-Encoder	Query 和 Doc 分别编码，计算相似度	速度快，可预计算	精度相对较低
Cross-Encoder	Query 和 Doc 拼接后联合编码	精度高	速度慢，无法预计算

3.4.2 两阶段检索策略

```
阶段1 (召回): Bi-Encoder 快速检索 Top-50
阶段2 (精排): Cross-Encoder 重排序得到 Top-5
```

```
class Reranker:
    def __init__(self, model_name="BAAI/bge-reranker-base"):
        self.model = CrossEncoder(model_name)

    def rerank(self, query: str, documents: List[str], top_k: int):
        # 构建 query-document 对
        pairs = [[query, doc] for doc in documents]

        # Cross-Encoder 打分
        scores = self.model.predict(pairs)

        # 按分数排序
        ranked = sorted(zip(documents, scores), key=lambda x: x[1],
reverse=True)
        return ranked[:top_k]
```

3.5 缓存机制

为避免重复的 LLM 调用（查询重写、HyDE 生成），实现 LRU 缓存：

```
class QueryCache:
    def __init__(self, max_size: int = 1000):
        self.cache = OrderedDict()
        self.max_size = max_size
        self.hits = 0
        self.misses = 0

    def get(self, key: str):
        if key in self.cache:
            self.cache.move_to_end(key)  # LRU 更新
            self.hits += 1
            return self.cache[key]
        self.misses += 1
        return None

    def set(self, key: str, value):
        if len(self.cache) >= self.max_size:
            self.cache.popitem(last=False)  # 移除最旧项
```

```
        self.cache[key] = value
```

3.6 完整 RAG Pipeline

```python
def retrieve(
    self,
    query: str,
    top_k: int = 5,
    enable_rewrite: bool = True,
    rewrite_mode: str = "single",
    enable_rerank: bool = True
):
    # 1. 查询重写/HyDE
    if enable_rewrite:
        if rewrite_mode == "hyde":
            hyde_doc = self.query_rewriter.generate_hyde_document(query)
            search_vector = self.embedder.embed([hyde_doc])[0]
        elif rewrite_mode == "multi":
            queries = self.query_rewriter.generate_multi_queries(query)
            return self._retrieve_multi(queries, top_k, enable_rerank)
        else:
            rewritten = self.query_rewriter.rewrite(query, mode=rewrite_mode)
            search_vector = self.embedder.embed([rewritten])[0]
    else:
        search_vector = self.embedder.embed([query])[0]

    # 2. 向量检索
    candidates = self.vector_store.search(search_vector, top_k=top_k * 3)

    # 3. 重排序
    if enable_rerank and self.reranker:
        results = self.reranker.rerank(query, candidates, top_k)
    else:
        results = candidates[:top_k]

    return results
```

---

4. 实验结果

4.1 评估指标

指标	说明	计算方式
准确率 (Accuracy)	答案相对于标准答案或医学事实的正确程度。	正确回答数 / 总问题数
引用精确率 (Citation Precision)	生成内容中引用的参考文献片段是否真正支持了该处的陈述。	有效引用数 / 总引用数
引用召回率 (Citation Recall)	参考文献中与问题相关的关键信息是否被包含在生成答案中。	已引用相关信息 / 所有相关信息
引用 F1 (Citation F1)	引用精确率和引用召回率的调和平均数，综合反映引用质量。	2 * (P * R) / (P + R)
幻觉率 (Hallucination Rate)	模型生成的回答中包含未在参考文档中提及的虚假信息的比例。	包含幻觉的回答数 / 总回答数

4.2 实验配置对比

配置	查询重写	HyDE	重排序
Baseline	✗	✗	✗
+Rewrite	✓ (single)	✗	✗
+Multi	✓ (multi)	✗	✗
+HyDE	✗	✓	✗
+Rerank	✗	✗	✓
+Rewrite+Rerank	✓	✗	✓
+HyDE+Rerank	✗	✓	✓

4.3 检索效果对比

实验配置	准确率 (Accuracy)	引用精确率 (Precision)	引用召回率 (Recall)	引用 F1	幻觉率 (Hallucination)	耗时 (s)
Baseline (基础版)	70.02%	0.6913	0.5867	0.6347	2.42%	-
Optimized (优化版)	**76.00%**	**0.7470**	**0.6480**	**0.6940**	**2.00%**	100.5

5. 问题分析与创新点

5.1 遇到的问题与解决方案

问题 1：复杂多跳查询（Multi-hop）效果差

现象：当用户提出的问题需要跨文档推理时（例如“二型糖尿病的一线用药有哪些副作用？”需要先通过文档 A 确认“一线用药是二甲双胍”，再通过文档 B 查询“二甲双胍的副作用”），系统往往仅能基于语义相似度检索到其中一部分片段，无法构建完整的证据链，导致回答片面。

解决方案：

- **引入查询分解（Query Decomposition）**：利用 LLM 将复杂的复合问题拆解为多个简单的独立子查询（Sub-queries），分别进行并行或串行检索。但是时间会大量增加。
- **实现迭代检索（Iterative Retrieval）**：基于第一轮检索到的上下文，让模型判断信息是否充足，若不足则根据已知信息生成新的检索词进行二次检索。类似于self-RAG的形式

问题 2：多轮对话上下文丢失

现象：用户在对话中使用代词（"它"、"那个"），系统无法理解。

解决方案：

- 实现上下文感知的查询重写

5.2 创新点

创新点 1：多策略融合的查询优化

本系统集成了多种查询优化策略，可根据场景灵活选择：

策略	适用场景	优势
Single Rewrite	简单查询	低延迟，效果稳定
Multi Query	复杂/模糊查询	高召回率
Context Rewrite	多轮对话	上下文理解
HyDE	专业领域查询	弥合词汇鸿沟

创新点 2：两阶段检索架构

```
Bi-Encoder (快速召回) → Cross-Encoder (精确排序)
     ↓                          ↓
  Top-50 候选                Top-5 精排结果
```

平衡了检索速度和精度，适合大规模知识库。

创新点 3：可解释的答案生成

- 每个答案都附带参考来源
- 展示检索文档的相关度分数
- 支持追溯答案依据

创新点 4：完整的工程实现

- 前后端分离架构，易于部署和扩展
- 流式输出，提升用户体验
- 智能缓存，优化响应速度
- 多 LLM Provider 支持，灵活切换

---

6. Demo 演示

6.1 系统截图

6.1.1 主界面

6.1.2 对话示例

6.1.3 配置面板

6.1.4 来源引用展示

7. 未来改进方向

7.1 构建知识图谱

构建一个以知识图谱为核心的RAG，具体实现方法：
1、输入的文档用LLM提取一下实体与关系。然后让LLM描述一下实体和关系，作为实体与关系的键值

2、将这些是实体与关系插入到知识图谱中，可以使用neo4j。nebula在数据量大的时候适合，但是普通实验就算了

3、将知识图谱的实体、关系、还有二者的描述全部embedding进向量库，之后查询的，先查询一下向量库，这样子可以找到语义相似的节点。

4、查询的适合先重写一下，然后提取一下实体或者里面的一些关键字，然后查询向量库，把相似的取出来，去知识图谱里面查询，将结果给LLM，总结