臨床意思決定支援のための検索拡張生成 (RAG) システムの提案および実装

2581008　井上 裕介

2025 年 7 月 3 日

1　はじめに

(1)

本レポートは、臨床意思決定支援に特化した検索拡張生成 (RAG) システムの設計、実装、および評価について述べる。汎用大規模言語モデル (LLM) が、医療のような高リスク領域において、ハルシネーションや古い知識、文脈的根拠の欠如といった重大な限界を抱えていることに着目した。これらの限界は患者の安全に直接的な脅威となり、最適な医療提供の障害となる。この課題に対処するため、LLM を「閉じた本の試験」から「開いた本の試験」へと変革するアプローチである RAG が不可欠であると論じている。RAG は、ユーザーの質問に対し、外部の信頼できる知識ベース (電子カルテから構築された精選された知識ベース) から関連情報を検索し、それを LLM に提供することで、提示された証拠に基づく応答生成を強制する。この転換は、ハルシネーションを削減し、最新情報へのアクセスを可能にし、応答の追跡可能性と説明可能性を確保することで、エビデンスに基づく実践のパラダイムにシステムを近づける。本システムは、実世界の臨床ノートの代理として MIMIC-IV データセットを利用し、医学文書のセマンティックな完全性を維持するためのメタデータ拡張チャンキング戦略と、日本語に特化した医療用埋め込みモデルを活用して実装された。評価は RAGAS フレームワークを用いて行われ、臨床情報アクセスを強化する RAG の有効性が示されている。

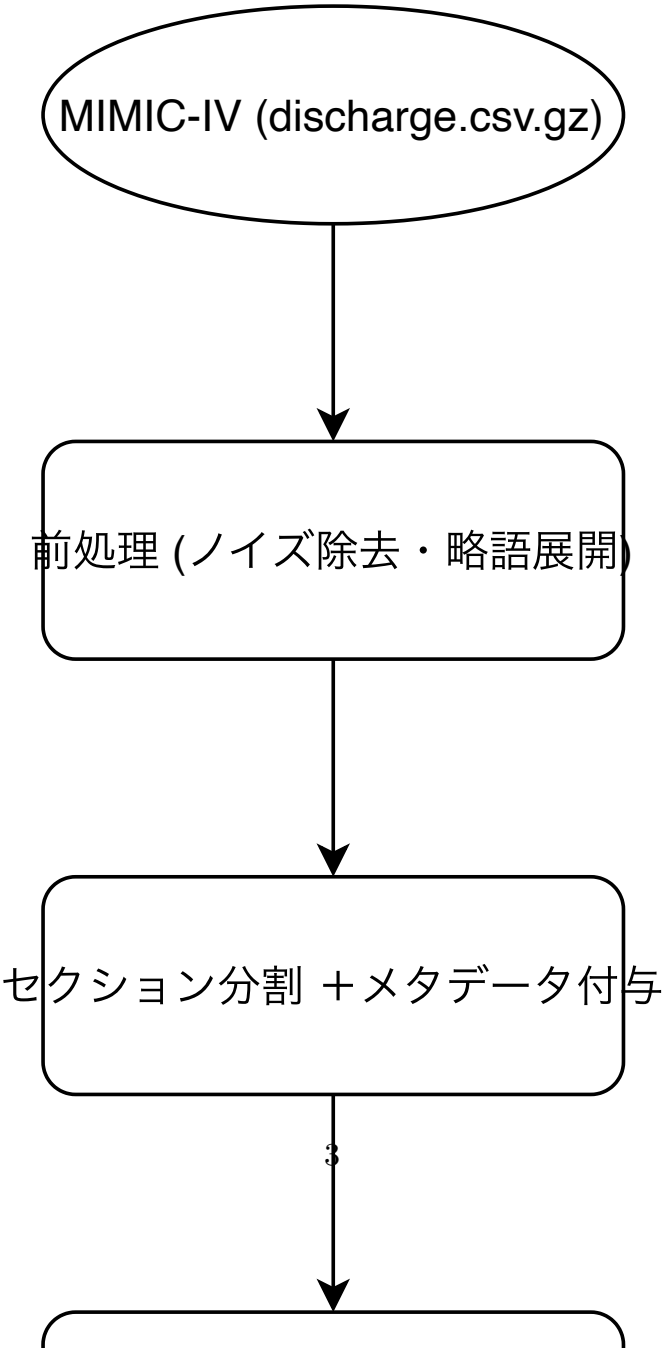
2 Retrieval-Augmented Generation

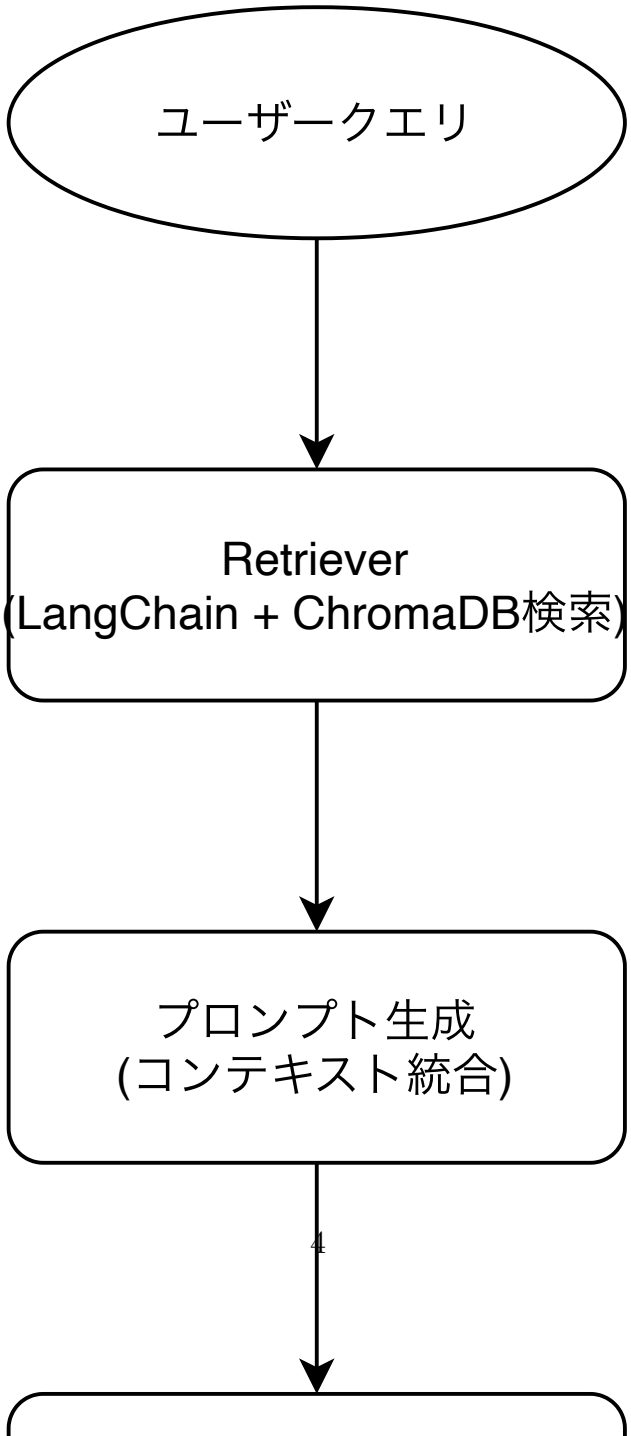
Retrieval-Augmented Generation (RAG) は、ユーザーの質問に対して外部の知識ベースから情報を検索し、それを基に応答を生成する手法である。RAG は、特に医療のような高リスク領域において、信頼性の高い情報を提供するために重要である。RAG の基本的な流れは以下の通りである。

1. ユーザーの質問を受け取る
2. 質問に関連する情報を外部の知識ベースから検索する
3. 検索結果を基に、LLM が応答を生成する
4. 応答をユーザーに提供する

RAG は、応答からソース文書への直接的な証拠の追跡を可能にし、これは現代医療の礎であるエビデンスに基づく実践 (Evidence-Based Practice) のパラダイムにシステムを近づけるものであるとされている。

3　システムの構成




ユーザークエリ
Retriever
(LangChain + ChromaDB検索)
プロンプト生成
(コンテキスト統合)

RAG システムの中核的なメカニズムは、ユーザーの質問（クエリ）がまず外部の信頼できる知識ベースからの検索ステップを引き起こすことから始まる。この検索されたコンテキスト（文脈）が、元のプロンプトに「拡張（augment）」されて LLM に提供され、LLM はこの与えられた特定の情報に基づいて応答を生成する。この構造により、LLM の応答生成は、確率的な推測から提示された証拠に基づく合成へと転換される。システムの設計においては、実世界の臨床ノートの代理として MIMIC-IV データセットが利用された。医学文書のセマンティックな完全性を維持するため、メタデータで拡張された高度なチャンキング戦略が採用されたアーキテクチャが設計されている。また、ドメインの関連性を確保するために、日本語に特化した医療用埋め込みモデルを活用している。

4 実装

ここからは、RAG システムの実装に関する詳細を述べる。本 RAG システムの構築と展開には多くの工数を必要とする。これらの作業は知識ベースの品質に決定的に依存する。そのため、文書の元の意味構造を保持し、メタデータでそれを豊かにする高度なインジェスチョンパイプラインへの投資が最もレバレッジの効く活動とされた。本章では、LangChain フレームワークを用いて、前章で設計したデータパイプラインと RAG のコアロジックを実装する具体的な Python コードと手法について解説する。

4.1 データパイプラインの設計と実装

はじめに、データ処理パイプラインは、MIMIC-IV-Note データセットの discharge.csv.gz ファイルをロードすることから始まる。実際の EHR データに含まれるノイズを処理するため、正規表現などを用いて以下のクリーニングを行う。

- 不要な空白や一貫性のないフォーマットの除去
- セクションヘッダーの標準化
- MIMIC-IV の匿名化処理で生じるアーティファクトの適切な処理

4.2 メタデータの拡張とセクションベースの解析

次に、セクションベースの解析について述べる。医療記録のような構造化された文書に対して、単純なチャンキング戦略は不十分である。文脈を不自然に分断し、検索されるチャンクの有用性を損なう可能性があるためである。そこで、本プロジェクトではよりイ

ンテリジェントなセクション認識型のアプローチを実装する。まず退院サマリーを構成セクションに解析することである。MIMIC ノートに関する研究によれば、これらは半構造化されたフォーマットを持ち、共通のヘッダーが存在する。正規表現を用いて、以下のようなヘッダーを特定し、その配下のテキストを抽出する。

- Chief Complaint (主訴)
- History of Present Illness (現病歴)
- Past Medical History (既往歴)
- Allergies (アレルギー)
- Medications on Admission (入院時処方薬)
- Physical Exam (身体所見)
- Brief Hospital Course (入院経過概要)
- Discharge Diagnosis (退院時診断)
- Discharge Medications (退院時処方薬)

このセクションベースの解析により、各セクションが独立したコンテキストとして扱われ、検索精度が向上する。

4.3 メタデータの定義と埋め込みモデルの選定

次に、メタデータの定義と埋め込みモデルの選定について述べる。メタデータは、各セクションのコンテキストを豊かにし、検索精度を向上させるための重要な要素である。最終的に ChromaDB の各ベクトルと共に格納される、リッチなメタデータ構造を以下のように定義する。

Listing 1　提案スキーマ (JSON 形式)

```
{
  "source_file": "discharge_12345.txt",
  "note_id": "note_xyz",
  "patient_id": "p_001",
  "admission_id": "adm_001",
  "note_type": "Discharge␣Summary",
  "section_header": "Discharge␣Medications",
  "chunk_id": "note_xyz_chunk_5",
  "snomed_ct_codes": ["73211009", "386661006"],
  "icd10_codes": ["E11.9", "I10"]
}
```

このメタデータは単なる記述情報ではなく、検索精度を向上させるための強力なフィルタリングツールとして機能する。このスキーマは、強力なハイブリッド検索戦略を可能に

する。例えば、臨床医が「入院 ID adm_001 の患者の退院時処方薬は何でしたか？」と質問した場合、システムはまずベクトル空間全体から admission_id == ”adm_001” かつ section_header == ”Discharge Medications” であるチャンクをフィルタリングし、その非常に小さく関連性の高いサブセット内でクエリのセマンティック検索を実行できる。これは、データベース全体に対するブルートフォース検索よりもはるかに効率的かつ正確である。また、SNOMED-CT や ICD-10 コードをメタデータに含めることで、コードベースの検索や、オントロジーを利用したより高度なクエリへの拡張性も確保できる。

4.4 埋め込みモデルのロード

まず、選定した日本語医療用埋め込みモデル kasys/jmed-me5-v0.1 をロードする。このモデルは最適な性能を発揮するために、クエリと文書で異なるプレフィックスを要求するため、その設定も行う。

Listing 2 埋め込みモデルのロード

```
from langchain_huggingface import HuggingFaceEmbeddings

# モデル名とプレフィックス設定
model_name = "kasys/jmed-me5-v0.1"
model_kwargs = {'device': 'cpu'} # or 'cuda'
encode_kwargs = {'normalize_embeddings': True}

# プレフィックスはモデルの性能に重要
# クエリ用と文書用で異なる設定を行う
hf_embeddings = HuggingFaceEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs,
    query_instruction="query: ", # クエリ用のプレフィックス
    embed_instruction="passage: " # 文書用のプレフィックス
)
```

4.5 ChromaDB の初期化とインデックス作成

次に、作成したベクトルを永続的に保存するため、ChromaDB クライアントを初期化し、前処理済みの文書をインデックスに追加する。ID 生成ロジックは、再登録時に重複を防ぎ、更新を可能にするために重要である。

Listing 3 ChromaDB の初期化とインデックス作成

```
import chromadb
from langchain_chroma import Chroma
import os

```

```
# 永続化ディレクトリを指定
persist_directory = "./chromadb_store"
collection_name = "medical_notes_jp"

# クライアントを初期化ChromaDB
client = chromadb.PersistentClient(path=persist_directory)

# のラッパーを初期化LangChainChroma
vector_store = Chroma(
    client=client,
    collection_name=collection_name,
    embedding_function=hf_embeddings,
)

# は第章で作成したオブジェクトのリストpreprocessed_documents4Document
# 各はとを持つDocumentpage_contentmetadata

# ユニークのリストを作成ID
# 例: metadata['sourceと']metadata['start_indexを組み合わせる']
ids = [
    f"{os.path.splitext(doc.metadata['source'])}_{doc.metadata['start_index']}"
    for doc in preprocessed_documents
]

# ドキュメントをに追加（動作）ChromaDBUPSERT
vector_store.add_documents(documents=preprocessed_documents, ids=ids)
```

4.6 LangChain のリトリーバの設定

作成した vector_store を LangChain のリトリーバとして設定する。ここでは、先ほど設計したメタデータによるフィルタリングを活用した検索も可能である。

Listing 4 LangChain のリトリーバの設定

```
retriever = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={'k': 5} # 上位件のチャンクを検索5
)

# メタデータフィルタリングを用いたリトリーバの例
# 例：特定の患者の「退院時処方薬」セクションのみを検索対象とする
metadata_filtered_retriever = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={
        'k': 5,
        'filter': {
            "patient_id": "p_001",
            "section_header": "Discharge Medications"
        }
    }
)
```

4.7　RAG システムの実装

LangChain Expression Language (LCEL) を用いて、検索から生成までの一連の処理をパイプとして結合する。RunnablePassthrough は、元の質問をチェーンの後のステップに渡すために使用される。

Listing 5　RAG チェーンの構築

```
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

# の初期化（第章で選定）LLM3
llm = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)

# プロンプトテンプレート（次節で詳述）
prompt = hub.pull("rlm/rag-prompt") # またはカスタムプロンプト

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

# チェーンの定義RAG
rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
| prompt
| llm
| StrOutputParser()
)

# チェーンの実行
question = "患者の退院時処方薬について教えてください。p_001"
response = rag_chain.invoke(question)
print(response)
```

4.8　プロンプトテンプレートの設計

LLM を安全かつ正確に導くためには、精巧に設計されたプロンプトが不可欠である。以下に、本プロジェクトで採用するプロンプトテンプレートとその設計思想を示す。

Listing 6　RAG プロンプトテンプレート

```
"あなたは高度なスキルを持つ臨床意思決定支援アシスタントです。あなたの目的は、提供された患者の電子カルテのコンテキストに「のみ」基づいて、質問に正確に回答することです。あなた自身の内部知識は一切使用しないでください。"
"もし提供されたコンテキストに質問に答えるために必要な情報が含まれていない場合は、必ず「提供されたコンテキストには、この質問に回答するための十分な情報が含まれていません。」と回答してください。存在しない情報を推測したり、推論したりしようとしないでください。"

"コンテキスト:"

{context}

"上記のコンテキストに基づき、以下の質問に回答してください。回答に使用する各情報について、具体的なソース文書とセクションを引用してください。"
```

```
9
10  "質問: {question}"
11
12  "回答:"
```