第九章　RAGアプリ

Chapter 5では、ローカル文書に基づいた質問応答システムを完成させました。現在の流行語で言えば、RAGアプリケーションを実装したことになります。

RAGとは何でしょうか？その正式名称はRetrieval-Augmented Generation（検索強化生成）で、検索と生成の能力を組み合わせたものです。RAGは、従来の言語生成モデルと大規模な外部知識ベースを組み合わせることで、モデルが応答やテキストを生成する際に、これらの知識ベースから動的に関連情報を検索できるようにします。この組み合わせ方法は、モデルの生成能力を強化し、特に具体的な詳細や外部事実のサポートが必要な場合に、より豊かで正確で根拠のあるコンテンツを生成することを目指しています。

第九章　RAGアプリ

➢　RAG の仕組み

RAGの動作原理は、以下のステップで概括できます。

検索：与えられた入力（質問）に対して、モデルはまず検索システムを使用して、大規模な文書コレクションから関連する文書や段落を探します。この検索システムは、通常、ChromaDBやFaissなどの密なベクトル検索に基づいています。

コンテキストエンコーディング：関連する文書や段落が見つかった後、モデルはそれらを元の入力（質問）と一緒にエンコードします。

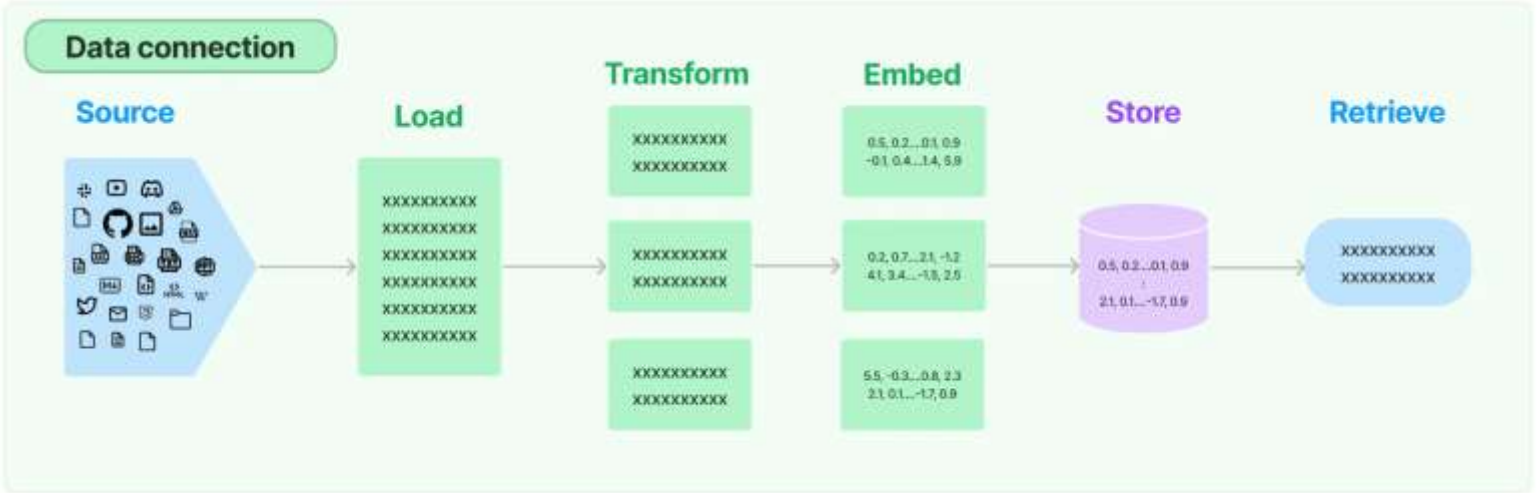
生成：エンコードされたコンテキスト情報を使用して、モデルは出力（回答）を生成します。これは通常、大規模なモデルを用いて行われます。

第九章　RAGアプリ

➢　RAG の仕組み

RAGの重要な特徴は、トレーニングデータ内の情報に依存するだけでなく、大規模な外部知識ベースから情報を検索できることです。これにより、RAGモデルはトレーニングデータに存在しない問題にも対応できるため、特に有用です。

RAGタイプのタスクは、現在の企業実際のアプリケーションシーンにおいて非常に需要が高く、LangChainもこの分野に重点を置いています。このセクションでは、LangChainにおけるRAG関連のすべてのツールを整理し、LangChainがこの分野でどのようなことができるかを把握できるようにします。



第九章　RAGアプリ

➢　文書のロード

RAGの第一歩は文書のロードです。LangChainは、さまざまなタイプの文書（HTML、PDF、コードなど）をロードするための複数の文書ローダーを提供しており、AirbyteやUnstructured.IOなどの主要なプロバイダーとも統合しています。

以下に、一般的な文書ローダーのリストを示します。

名前	説明	サンプルコード
TextLoader	テキスト文書を読み込む	from langchain.document_loaders import TextLoader loader = TextLoader("example_data/index.md") data = loader.load()
CSVLoader	CSV文書を読み込む	from langchain.document_loaders.csv_loader import CSVLoader loader = CSVLoader(file_path='example_data/abc.csv') data = loader.load()
HTMLLoader	HTML文書を読み込む	from langchain.document_loaders import UnstructuredHTMLLoader loader = UnstructuredHTMLLoader("example_data/content.html") data = loader.load()
JSONLoader	JSON文書を読み込む	from langchain.document_loaders import JSONLoader import json file_path='/example_data/chat.json' data = json.loads(Path(file_path).read_text())
MarkdownLoader	Markdown文書を読み込む	from langchain.document_loaders import UnstructuredMarkdownLoader markdown_path = "/example_data/README.md" loader = UnstructuredMarkdownLoader(markdown_path) data = loader.load()
PDFLoader	PDF文書を読み込む	from langchain.document_loaders import PyPDFLoader loader = PyPDFLoader("example_data/paper.pdf") pages = loader.load_and_split()

第九章　RAGアプリ

➢　テキスト変換

文書をロードした後、次のステップはテキスト変換です。最も一般的なテキスト変換は、長い文書をより小さなブロック（チャンクやノード）に分割し、モデルのコンテキストウィンドウに適合させることです。LangChainには、多くの組み込みの文書変換ツールがあり、文書を簡単に分割、結合、フィルタリング、または他の方法で操作することができます。

➢　テキスト分割器

長いテキストをブロックに分割することは一見簡単に思えますが、実際にはいくつかの細かい点があります。テキスト分割の質は、検索結果の質にも影響を与えます。理想的には、意味的に関連するテキストフラグメントを一緒に保つことが望ましいです。

第九章　RAGアプリ

➢　テキスト分割器

LangChainでのテキスト分割器の動作原理は以下の通りです：

1. テキストを小さく、意味的に関連するブロック（通常は文）に分割します。
2. これらの小さなブロックを組み合わせて、一定の大きさに達するまで大きなブロックを作成します。
3. 指定された大きさに達すると、1つのブロックが形成され、新しいテキストブロックの作成が始まります。この新しいブロックと直前のブロックの間には、一部が重なるようにして文脈を保ちます。

そのため、LangChainが提供するさまざまなテキスト分割器は、以下の観点から分割戦略とパラメータを設定するのに役立ちます：

- テキストの分割方法
- ブロックのサイズ
- ブロック間の重なり部分の長さ

第九章　RAGアプリ

➢　テキスト分割器

これらのテキスト分割器の説明と例は以下の通りです：

名前	説明
文字分割器 (CharacterTextSplitter)	単一文字で分割を行う。デフォルトの区切り文字は ”/n/n"。チャンクサイズは文字数に基づいて測定される。
再帰的文字分割器 (RecursiveCharacterTextSplitter)	文字列のリストを使って分割し、リスト内の順番に基づいてチャンクを分割する。分割するテキストのサイズが限界に達するまで続ける。デフォルトの区切りリストは [”/n/n‟, ”/n‟, ” ”, ””]
Markdown 見出し分割器 (MarkdownHeaderTextSplitter)	指定した見出しに基づいて Markdown ファイルを分割する。例として、「#」や「##」に基づいて分割が可能。
トークン分割器 (Token Splitter)	言語モデルにはトークン制限があるため、一連の分割器があり、分割時にトークン数を考慮する必要がある。

第九章　RAGアプリ

➢　テキスト分割器

テキスト分割を実践する際に考慮すべき具体的な要素について、以下の点をまとめました。

まず、LLMの具体的な制限についてです。GPT-3.5-turboは4096トークンのコンテキストウィンドウをサポートしており、これは入力トークンと生成された出力トークンの合計が4096を超えないようにする必要があることを意味します。制限を超えないようにするためには、入力プロンプトとして約2000トークンを予約し、返されるメッセージのために約2000トークンを残すことができます。これにより、5つの関連情報ブロックを抽出する場合、各ブロックのサイズは400トークンを超えないようにする必要があります。

第九章　RAGアプリ

➢　テキスト分割器

また、テキスト分割の戦略はタスクの種類に関連しています。

- 詳細なテキスト分析が必要なタスクでは、より小さなブロックを使用するのが良いでしょう。例えば、スペルチェック、文法チェック、テキスト分析は、テキスト内の単語や文字を識別する必要があります。また、スパムフィルタリング、盗用検出、感情分析、検索エンジン最適化（SEO）、トピックモデルで使用されるキーワード抽出などもこのような詳細なタスクに該当します。

- テキストの全体像を把握するタスクでは、より大きなブロックを使用します。例えば、機械翻訳、テキスト要約、質問応答タスクはテキストの全体的な意味を理解する必要があります。また、自然言語推論、質問応答、機械翻訳ではテキスト内の異なる部分間の関係を認識する必要があります。創作に関わるタスクもこのカテゴリーに含まれます。

第九章　RAGアプリ

➢　テキスト分割器

最後に、分割するテキストの性質も考慮する必要があります。例えば、テキストの構造が強い場合（コードやHTMLなど）は、より大きなブロックを使用するのが良いでしょう。一方、テキストの構造が弱い場合（小説やニュース記事など）は、より小さなブロックを使用する方が適しています。
異なるサイズのブロックやブロック間の重なりウィンドウのサイズを繰り返し試すことで、特定の問題に最適な解決策を見つけることができます。

第九章　RAGアプリ

➢　その他の形式のテキスト変換

テキスト分割以外にも、LangChainは文書に対するさまざまな種類の変換を行うためのツールを統合しています。以下、それらを順に分析します。

冗長な文書のフィルタリング：EmbeddingsRedundantFilter ツールを使用して、類似する文書を識別し、冗長な情報をフィルタリングします。これは、多くの類似またはほぼ同一の文書がある場合に、それらの余分なコピーを識別して削除し、ストレージの節約や検索効率の向上に役立ちます。

文書の翻訳：ツール doctran との統合を通じて、文書をある言語から別の言語に翻訳できます。

メタデータの抽出：doctran を使用して、文書の内容から日付、著者、キーワードなどの重要な情報を抽出し、メタデータとして保存できます。メタデータは文書の属性や内容を説明するデータであり、文書の管理、分類、検索をより効率的に行えます。

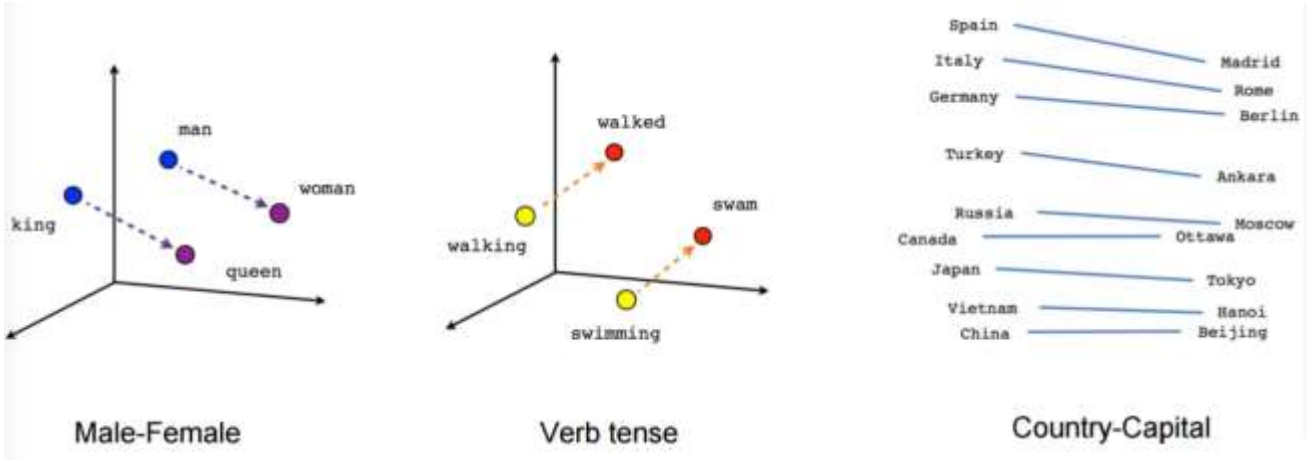
このように、文書変換は単なるテキスト分割にとどまらず、より良いインデックス作成と検索機能を実現するために文書を最適化する追加の操作も含まれます。

第九章　RAGアプリ

➢　テキストのEmbeddings

テキストブロックが形成された後、次にLLMを使ってEmbeddingsを行い(第三章で紹介しました)、テキストを数値表現に変換します。これにより、コンピュータがテキストをより容易に処理・比較できるようになります。OpenAI、Cohere、Hugging Faceには、テキストのEmbeddingを行うモデルがあります。

Embedding はテキストのベクトル表現を作成し、これによってベクトル空間でテキストを扱い、**意味検索（Semantic Search）**などの操作を実行できます。つまり、ベクトル空間内で最も類似したテキスト片を検索することが可能です。



第九章　RAGアプリ

➢　テキストのEmbeddings

LangChainの**Embeddingsクラス**は、テキストモデルとのやり取りを行うために設計されたクラスです。このクラスは、これらすべての提供者に対して標準的なインターフェースを提供します。

```
from langchain.embeddings import OpenAIEmbeddings

embeddings_model = OpenAIEmbeddings()
```

このクラスは2つのメソッドを提供します：

embed_documents メソッド：文書に対してEmbeddingを作成します。このメソッドは複数のテキストを入力として受け取り、複数の文書を一度にベクトル表現に変換することができます。

embed_query メソッド：クエリ（検索問い合わせ）に対してEmbeddingを作成します。このメソッドは1つのテキスト（通常はユーザーの検索クエリ）を入力として受け取ります。

第九章　RAGアプリ

➢　テキストのEmbeddings

なぜ2つの方法が必要なのでしょうか？一見すると、これらの2つの方法はどちらもテキスト埋め込みのためのものに見えますが、LangChainではそれらを分けています。その理由は、いくつかの埋め込み提供者が文書とクエリに異なる埋め込み方法を使用しているためです。文書は検索対象の内容であり、クエリは実際の検索リクエストです。これら2つはその性質や目的によって、異なる処理や最適化が必要となる場合があります。

embed_documents メソッドとembed_query メソッドの例コードは以下の通りです：

```
embeddings = embeddings_model.embed_documents(
    [
        （会話の内容）
    ]
)
```

```
embedded_query = embeddings_model.embed_query("お問い合わせ内容")
```

第九章　RAGアプリ

➢　Embeddingsの保存

Embeddingsの計算は時間がかかる場合があります。このプロセスを加速するために、計算済みのEmbeddingsを保存または一時的にキャッシュすることができます。これにより、次回それらが必要になった際には再計算せずに直接読み込むことができます。

CacheBackedEmbeddings は、キャッシュ機能をサポートするEmbeddingsラッパーであり、Embeddingsをキー・バリュー型ストレージにキャッシュすることができます。具体的な操作は、テキストをハッシュ処理し、このハッシュ値をキャッシュのキーとして使用することです。CacheBackedEmbeddingsを初期化する主な方法は、from_bytes_store を使用することです。これには以下のパラメーターが必要です：

underlying_embedder：実際に計算するEmbedding器。
document_embedding_cache： Embeddingsを保存するためのキャッシュ。
namespace（オプション）：文書キャッシュのための名前空間で、他のキャッシュと衝突しないようにするため。

第九章　RAGアプリ

➢ Embeddingsの保存

異なるキャッシュ戦略は以下の通りです：

InMemoryStore：メモリ内でEmbeddingsをキャッシュします。主にユニットテストやプロトタイプ設計で使用されます。長期的に保存する必要がある場合は、このキャッシュを使用しないでください。

LocalFileStore：ローカルファイルシステムに保存します。外部データベースやストレージソリューションに依存したくない場合に適しています。

RedisStore：Redisデータベースにキャッシュします。高速かつスケーラブルなキャッシュソリューションが必要な場合に適した選択肢です。

第九章　RAGアプリ

➢ Embeddingsの保存

メモリ内で埋め込みをキャッシュする例コードは以下の通りです：

```
from langchain.storage import InMemoryStore

store = InMemoryStore()

from langchain.embeddings import OpenAIEmbeddings, CacheBackedEmbeddings

underlying_embeddings = OpenAIEmbeddings()

embedder = CacheBackedEmbeddings.from_bytes_store(
    underlying_embeddings,
    store,
    namespace=underlying_embeddings.model
)

embeddings = embedder.embed_documents([“こんにちは”, “さよなら"])
```

第九章　RAGアプリ

➢　Embeddingsの保存

コードの説明をします。まず、メモリ内にストレージスペースを設定し、その後、実際にEmbeddingsを生成するツールを初期化します。それから、このツールはキャッシングツールでラップされ、2つのテキストに対してEmbeddingsを生成するために使用されます。

他の2種類のキャッシャーについても、使用方法はそれほど複雑ではありません。LangChainのドキュメントを参考にして、独自に学習することができます。

より一般的なベクトルの保存方法は、ベクトルデータベース（Vector Store）を使用して保存することです。LangChainは多くのベクトルデータベースをサポートしており、その中にはオープンソースのものもあれば、商用のものもあります。例えば、Elasticsearch、Faiss、Chroma、Qdrantなどがあります。

第九章　RAGアプリ

➢　検索

LangChainでは、Retriever、つまりリトリーバーがデータ検索モジュールの中心的な入り口です。リトリーバーは、非構造化クエリを通じて関連するドキュメントを返します。

ベクトルストアリトリーバーは最も一般的で、主にベクトル検索をサポートします。当然、LangChainには他の種類のストレージフォーマットをサポートするリトリーバーもあります。

以下に、エンドツーエンドのデータ検索機能を実装します。VectorstoreIndexCreatorを使用してインデックスを作成し、インデックスのqueryメソッド内で、vectorstoreクラスのas_retrieverメソッドを通じて、ベクトルデータベース（Vector Store）を直接リトリーバーとして使用し、検索タスクを完了します。

第九章　RAGアプリ

➢ 検索

コードはこちらです。

```
import os
os.environ["OPENAI_API_KEY"] = 'あなたのOpenAI API key'

from langchain.document_loaders import TextLoader
loader = TextLoader('対象ドキュメントへのPath', encoding='utf8')

from langchain.indexes import VectorstoreIndexCreator
index = VectorstoreIndexCreator().from_loaders([loader])

query = "質問の内容"
result = index.query(query)

print(result)
```

第九章　RAGアプリ

➢　検索

このデータ検索プロセスは非常に簡単です。これはLangChainの強力なラッピング機能のおかげです。vectorstore.pyにあるVectorstoreIndexCreatorクラスのコードを見てみると、vectorstore、embedding、text_splitter、さらにはdocument loader（もしfrom_documentsメソッドを使用している場合）までがラップされていることがわかります。

したがって、上記の検索機能は、前の章で説明した一連のツールの統合に相当します。また、以下のコードを使用して、インデックス作成器のvectorstore、embedding、text_splitterを明示的に指定し、必要に応じて他のベクトルデータベースや別のEmbeddingモデルなどに置き換えることもできます。

第九章　RAGアプリ

➢ Index

このセクションの最後に、LangChainにおけるインデックス（Index）について見てみましょう。簡単に言うと、インデックスは文書情報を効率的に管理し、位置を特定する方法であり、各文書にユニークな識別子を付与し、検索を容易にします。

Chapter 5の例では、インデックスを明示的に使用することなくRAGタスクを完了しましたが、複雑な情報検索タスクでは、文書を効果的に管理しインデックスを作成することが重要なステップです。LangChainが提供するインデックスAPIは、開発者にとって効率的で直感的な解決策を提供します。具体的には、以下の利点があります：

重複内容の回避：ベクトルストア内に冗長なデータが存在しないようにします。

変更された内容のみの更新：どの内容が更新されたかを検出し、不必要な再書き込みを避けます。

時間とコストの節約：変更されていない内容の埋め込みを再計算せず、計算リソースの消費を削減します。

検索結果の最適化：重複や関連性のないデータを減らし、検索の正確性を向上させます。

第九章　RAGアプリ

➢ Index

LangChainは、記録管理者（RecordManager）を利用して、どの文書がベクトルストアに書き込まれたかを追跡します。

インデックス作成時には、APIが各文書にハッシュ処理を行い、各文書にユニークな識別子を付与します。このハッシュ値は文書の内容だけでなく、文書のメタデータも考慮に入れています。

ハッシュ処理が完了すると、以下の情報が記録管理者に保存されます：

文書ハッシュ：文書の内容とメタデータに基づいて計算されたユニークな識別子。
書き込み時間：文書がベクトルストアに追加された時刻。
ソースID：文書の元の出所を示すメタデータフィールド。

この方法により、文書が複数回変換または処理されても、その状態と出所を正確に追跡できるため、文書データが正しく管理され、インデックス付けされることが保証されます。

第九章　RAGアプリ

➢　まとめ

検索強化生成（RAG）を使用して非構造化データを保存し検索する最も一般的な方法は、これらの非構造化データにEmbeddingを行い、生成されたベクトルを保存することです。そして、クエリ時には検索したいテキストにもEmbeddingを行い、そのクエリと「最も類似している」ベクトルを検索します。ベクトルデータベースは、データの保存とベクトル検索の実行を担当します。

ご覧のとおり、RAGは実際には非構造化データの「地図」を作成します。ユーザーがクエリを送信すると、そのクエリもEmbeddingしまれ、アプリケーションはこの「地図」の中で最も一致する位置を探して、迅速かつ正確に情報を検索します。