|  |  |
| --- | --- |
| GEN AI LEARNING  **Complete Generative AI Learning – New Year Challenge** | WRITTEN BY: ALOY |

***Day – 17***

CORRECTIVE RAG & LANG-GRAPH WORKFLOW

Corrective RAG (CRAG) enhances traditional RAG architecture by introducing validation and correction mechanisms into the retrieval process. The system operates through a sophisticated multi-stage workflow:

1. Initial Retrieval Stage
   * The system performs initial document retrieval from the knowledge base
   * Uses vector similarity to identify potentially relevant documents
   * Maintains context through overlapping document chunks
2. Document Validation Stage
   * Each retrieved document undergoes LLM-based evaluation
   * Generates confidence scores based on relevance to the query
   * Applies validation criteria to determine document usefulness
3. Adaptive Response Strategy
   * High-confidence documents: Proceeds to direct response generation
   * Medium-confidence documents: Initiates hybrid approach
   * Low-confidence documents: Triggers web search enhancement
4. Query Enhancement
   * Automatically reformulates queries when initial results are insufficient
   * Optimizes search terms for improved document retrieval
   * Maintains semantic alignment with original user intent

This implementation creates a production-ready Corrective RAG system that brings together several powerful components:

1. **Smart Document Assessment**:
   * Configurable confidence thresholds (0.8 for high confidence, 0.5 for medium)
   * Automated relevance scoring of retrieved documents
   * Dynamic decision-making based on document quality
2. **Models and Database**:
   * Claude 3.5 Sonnet for document evaluation and response generation
   * OpenAI embeddings for semantic search
   * Qdrant vector store for efficient document retrieval
3. **Adaptive Search Capabilities**:
   * Automatic query reformation when needed
   * Integrated Tavily API for web search fallback
4. **Interactive Interface**:
   * Streamlit-based user interface for easy testing
   * Real-time visibility into the RAG pipeline
   * Step-by-step workflow monitoring

**Note on LangGraph Workflow**

The **LangGraph workflow** in this code represents a structured way to execute a **retrieval-augmented generation (RAG)** process using graph-based task orchestration. The workflow leverages **LangGraph's StateGraph class** to define a directed graph where each node performs a specific function. These nodes are connected through edges, which determine the sequence of operations and enable conditional branching based on intermediate results.

**Key Concepts of LangGraph Workflow**

1. **StateGraph**:
   * A StateGraph represents a workflow composed of tasks (nodes) connected in a directed graph.
   * Each task processes the state (a TypedDict called GraphState in this case), modifies it, and passes it to the next node.
2. **Nodes**:
   * Nodes are discrete functions performing specific tasks like document retrieval, query transformation, grading relevance, or generating answers.
   * Each node takes the current state as input and returns a modified state.
3. **Edges**:
   * Edges define how nodes are connected and the order in which tasks are executed.
   * Conditional edges allow branching based on runtime decisions.
4. **Workflow Compilation**:
   * The StateGraph is compiled into an executable workflow (app), which processes user input and produces output.

**LangGraph Workflow Overview**

**Workflow Initialization**

* **Entry Point**:
  + The graph starts with the retrieve node, which retrieves relevant documents based on the user query.
* **State Initialization**:
  + The state is initialized with keys such as:
    - question: User-provided query.
    - documents: List of documents (updated by nodes as the workflow progresses).

**Defined Nodes**

1. **retrieve**:
   * Retrieves documents from a **Qdrant vector database** using similarity search on embeddings.
   * Inputs:
     + User's question (in the state).
   * Outputs:
     + Relevant documents appended to the state.
2. **grade\_documents**:
   * Evaluates the relevance of the retrieved documents using an LLM.
   * Filters out irrelevant documents.
   * Adds a run\_web\_search key to the state, indicating whether to perform a web search if documents are irrelevant.
3. **transform\_query**:
   * Refines or optimizes the user's query for better search results.
   * Ensures the query is suitable for web search engines or APIs like Tavily.
   * Outputs an updated question in the state.
4. **web\_search**:
   * Uses the **Tavily API** to fetch additional context if documents are insufficient.
   * Adds the web search results to the documents key in the state.
5. **generate**:
   * Generates an answer to the user query using an LLM (e.g., GPT-4 or Claude).
   * Takes into account the most relevant documents (retrieved or searched).
   * Outputs the final answer (generation key in the state).

**Conditional Flow**

The workflow includes conditional edges to decide the next step based on intermediate results:

* **Decision Function** (decide\_to\_generate):
  + If retrieved documents are irrelevant (run\_web\_search = "Yes"), it redirects to the transform\_query node to improve the query and perform a web search.
  + Otherwise, it directly proceeds to the generate node to create an answer.

**Graph Structure**

The workflow is structured as follows:

1. **Start**: retrieve
2. → grade\_documents
   * If relevant: → generate
   * If irrelevant: → transform\_query
     + → web\_search
       - → generate
3. **End**: generate

**Advantages of LangGraph Workflow**

1. **Modular Design**:
   * Tasks are encapsulated in reusable and independent nodes.
   * Nodes can be updated or replaced without affecting the overall workflow.
2. **Conditional Branching**:
   * Decisions are made dynamically based on the state, improving efficiency and adaptability.
3. **State Management**:
   * The state is passed and modified throughout the workflow, ensuring that data is consistently updated and accessible.
4. **Extensibility**:
   * New nodes or edges can be added to handle additional tasks or improve functionality.
   * For example, adding a sentiment analysis or summarization node before generating the final answer.
5. **Error Handling**:
   * Retry mechanisms (e.g., tenacity) are integrated into nodes like web search to ensure robustness.

**How LangGraph Supports RAG**

* **RAG Workflow**:
  + Combines information retrieval (from a vector database or web search) with generative AI to produce answers grounded in reliable context.
* **LangGraph Features**:
  + Provides a structured framework for executing the retrieval and generation steps in sequence or parallel.
  + Allows query refinement and context augmentation dynamically based on intermediate results.

**Practical Use Case**

* **Input**:
  + A user inputs a query like: "What are the experiment results in the research paper?"
* **Execution**:
  + Documents are retrieved from a Qdrant vector database.
  + The relevance of the retrieved documents is graded.
  + If necessary, a web search is performed to gather additional information.
  + The final response is generated based on all available context.
* **Output**:
  + A well-grounded and context-aware answer is displayed to the user.

This **LangGraph workflow** provides a systematic, scalable, and robust framework for building complex RAG pipelines, making it suitable for use cases like research, knowledge management, and conversational AI systems.

Code Explanation:

This code sets up a **Streamlit-based application** to enable a question-answering system using a **retrieval-augmented generation (RAG)** workflow. Below is an explanation of its components and their interactions:

**Key Components and Their Purpose**

1. **Imports and Dependencies**
   * It uses multiple libraries, such as:
     + **langchain**: For managing workflows, retrieval, document processing, embeddings, and prompts.
     + **streamlit**: To build the web application interface.
     + **qdrant\_client**: For vector database management to store and query document embeddings.
     + **pydantic**: For schema validation and data management.
     + **tenacity**: To implement retry mechanisms for resilient API calls.
2. **State Management**
   * **Session state variables** (st.session\_state) are used to store and manage API keys and configuration details (e.g., OpenAI, Tavily, Anthropic keys, Qdrant URL).
   * Functions:
     + initialize\_session\_state: Ensures default values for session variables.
     + setup\_sidebar: Creates a user-friendly sidebar for configuring the API keys and URLs.
3. **Document Loading**
   * The system can load documents from:
     + A URL (e.g., PDFs or webpages via WebBaseLoader).
     + Uploaded files (e.g., .pdf, .txt).
   * Uses loaders like PyPDFLoader, TextLoader, and WebBaseLoader based on file type or URL.
4. **Text Splitting and Vector Storage**
   * Documents are split into manageable chunks using RecursiveCharacterTextSplitter.
   * These chunks are embedded using OpenAI's embedding model and stored in a **Qdrant vector database** for similarity search.
5. **Workflow Definition**
   * The app defines a graph-based workflow (StateGraph) with nodes representing distinct tasks:
     + **Retrieve**: Retrieve relevant documents from the vector database.
     + **Grade Documents**: Check the relevance of the retrieved documents.
     + **Generate**: Use an LLM (e.g., OpenAI's gpt-4o or Anthropic’s Claude) to answer the user's question based on the retrieved context.
     + **Transform Query**: Refine the user's query for better search results.
     + **Web Search**: Search the web for additional context if needed (using the Tavily API).
6. **Integration of LangGraph**
   * Uses langgraph to define and manage the workflow logic:
     + **Conditional Nodes**: The workflow decides whether to generate a response directly or refine the query and perform a web search based on document relevance.
7. **Error Handling and Resilience**
   * Retries are implemented with the tenacity library to handle transient failures during web search or document retrieval.
8. **User Interface**
   * Users can input:
     + API keys and configuration details in the sidebar.
     + A document URL or upload a file.
     + Their query/question to be answered by the system.
   * The interface displays intermediate steps of the workflow (e.g., retrieved documents, graded relevance) and the final answer.

**Workflow Execution**

When the user inputs a question:

1. **Document Retrieval**:
   * Documents from the vector store are retrieved based on the question embeddings.
2. **Document Grading**:
   * The retrieved documents are assessed for relevance using an LLM.
3. **Conditional Path**:
   * If documents are irrelevant, the system refines the query (transform\_query) and performs a web search.
   * If documents are relevant, it proceeds to response generation.
4. **Answer Generation**:
   * Using an LLM, the system generates an answer based on the retrieved or newly searched context.
5. **Display**:
   * The workflow steps and final output are presented interactively in the Streamlit app.

**Unique Features**

1. **Hybrid Workflow**:
   * Combines retrieval-augmented generation (RAG) with web search and query refinement for robust performance.
2. **Dynamic Document Handling**:
   * Handles both static documents (e.g., PDFs) and live data from the web.
3. **Interactive User Interface**:
   * Enables users to see intermediate workflow outputs and adjust configurations.

**Use Case**

This application is ideal for **researchers, analysts, or any knowledge workers** needing to extract and synthesize insights from large text datasets, documents, or web content.

