Financial RAG Prototype

Development Methodology for Financial RAG Prototype

Overview

This documentation outlines the comprehensive development methodology for a Retrieval-Augmented Generation (RAG) system specifically designed to handle financial documents. The system leverages Python and Langchain to adeptly parse, process, and retrieve pertinent information from financial documents. Through extensive testing, the combination of LlamaParse for document parsing and Chroma as the vector store has proven to be the most accurate and reliable.

Technology Stack

- **Programming Language**: Python
- **Backend Framework**: Langchain
- **Document Parsers/Loaders**: LlamaParse
- **Vector Stores**: ChromaDB
- **Frontend Framework**: Streamlit

How RAG Works for Financial Documents

The Retrieval-Augmented Generation (RAG) system is designed to handle the complexity and specificity of financial documents. Here's a high-level step-by-step explanation of how RAG is implemented for financial documents:

1. Document Ingestion and Parsing

Step 1.1: Document Collection

- Collect financial documents such as annual reports, quarterly earnings, balance sheets, and income statements.

Step 1.2: Document Parsing Using LlamaParse

- Utilize LlamaParse to accurately parse these documents. LlamaParse is chosen for its ability to handle the intricate details of financial documents.
- Extract structured data, including text, tables, and figures, converting them into machine-readable formats.

2. Embedding Generation

Step 2.1: Text Embeddings

- Generate embeddings for the parsed content using a pre-trained language model. Embeddings transform text into numerical vectors that capture semantic meaning.

3. Vector Store Configuration

Step 3.1: Storing Embeddings in Chroma DB

- Store the generated embeddings in Chroma DB, an open-source vector store optimized for efficient data retrieval.

4. Query Processing

Step 4.1: User Query Input

- Accept user queries related to financial documents. The queries are transformed into embeddings to match them with stored document embeddings.

Step 4.2: Retrieving Relevant Documents

- Retrieve the most relevant documents or document segments from Chroma based on the query embedding.

5. Response Generation

Step 5.1: Generative Model Integration

- Integrate a generative model to create responses based on the retrieved documents. This model combines the retrieved information to generate a coherent answer.

Step 5.2: Formatting the Response

- Format the response to ensure it is clear and informative, including references to specific sections of the financial documents if necessary.

6. Feedback and Fine-Tuning

Step 6.1: User Feedback Collection

- Collect feedback from users on the accuracy and usefulness of the responses. This feedback decides whether the response should be regenerated.

By following these detailed steps, the RAG system can efficiently process and retrieve information from financial documents, providing accurate and relevant responses to user queries.

Development Steps

1. Initial Setup

The initial setup involves the installation of all necessary libraries and frameworks required to build the RAG system. This is a fundamental step to ensure that all dependencies are correctly installed.

2. Document Parsing

Document parsing is a crucial step in the development of the RAG system. It involves using LlamaParse to accurately parse financial documents. LlamaParse was selected after rigorous testing due to its superior performance in accurately handling the intricacies of financial documents.

3. Vector Store Configuration

Once the documents are parsed, the next step is to configure the vector store. Chroma has been chosen as the vector store for its efficiency in data retrieval. The vector store plays a pivotal role in storing document embeddings, which are essential for the RAG system to function effectively.

4. Integration with Langchain

Integrating the parsed documents and the configured vector store with Langchain is essential to enable the RAG capabilities. Langchain facilitates the interaction between the vector store and the RAG model, allowing for seamless query processing and information retrieval.

5. Testing and Validation

Testing and validation are critical to ensuring the RAG system's reliability. A series of questions related to financial documents are used to test the system. The responses are evaluated to verify the system's accuracy.

```
test_questions = [
    "What is the net income for Q1 2023?",
```

```
    "Summarize the financial performance for the year 202
2."
]

for question in test_questions:
    response = rag_model.query(question)
    print(f"Question: {question}\\nResponse: {response}
\\n")
```

Remarks on Tried Methods

In the process of developing the RAG system, several document parsers/loaders and vector stores were tested. Below are the remarks on each method:

Langchain Compatible Document Loaders:

- **Azure Document Intelligence**: Works at-par with LlamaParse but is costlier.
- **Unstructured**: Handles text well to get a markdown output but tables seem to be a problem.
- **LlamaParse**: Demonstrated superior accuracy and reliability in parsing financial documents.

Langchain Compatible Vector Stores:

- **Azure AI Search Store**: Very costly.
- **FAISS**: Has a cap for a batch of 5+ queries/minute. This is too less, given our use case.
- **Chroma**: The most effective open-source vector store, providing efficient and accurate data retrieval.

Test Results

The following spreadsheet summarizes the results of the tests conducted on the RAG system. Each question was evaluated to determine if the system's response was accurate.

Documents Used: Annual 10-K filings for Microsoft, Apple, and Uber

`Test Plan - RAG Prototype.xlsx`

Conclusion

This detailed methodology ensures that the RAG system developed is robust, accurate, and reliable for processing financial documents. The combination of LlamaParse and Chroma has

been validated as the most effective setup, ensuring the system meets the stringent requirements of financial document analysis.

Next Steps

1. **Developing a User Chat Prototype Interface Using Streamlit**:
    - **Design and Layout**: Create an interactive chat interface using Streamlit that allows users to easily query the RAG system. Ensure the layout is intuitive, with clear input fields and response areas.
    - **Functionality**: Implement features like chat history, and stream response in real-time to enhance user experience.
    - **Feedback Mechanism**: Include a feedback mechanism for users to report issues or provide suggestions for improvement. User should be able to regenerate the response.
2. **Reducing Time for Response per Query**:
    - **Pipeline Optimization**: Analyze the current query processing pipeline and identify bottlenecks. Implement optimizations to reduce latency.
    - **Caching Mechanisms**: Investigate and implement caching mechanisms to store frequently accessed data, reducing the need for repeated processing.
    - **Algorithm Improvements**: Explore and implement more accurate, efficient algorithms for query processing and data retrieval.
3. **Creating a Job for Indexing Documents**:
    - **Automated Job Scheduling**: Develop a scheduled job to automate the indexing of new financial documents. Use reliable scheduling tools like Cron or Airflow.
    - **Incremental Indexing**: Implement incremental indexing to only process new or updated documents, reducing the time and resources required.
    - **Error Handling**: Develop robust error handling mechanisms to ensure the indexing job can recover from failures without manual intervention.
    - **Resource Allocation**: Ensure the job is efficient and does not impact the performance of the RAG system during regular operation by optimizing resource allocation.
4. **Reducing Cost by Optimizing Chunk Sizes and Chunk Retrieval**:
    - **Chunk Size Analysis**: Analyze and optimize the chunk sizes used for document parsing and storage. Ensure chunks are neither too large nor too small to balance

processing efficiency and retrieval speed.

- **Query Optimization**: Implement strategies to minimize retrieval costs by optimizing vector store queries. This could include refining query parameters or using more efficient indexing techniques.
- **Cost-Effective Storage Solutions**: Investigate and utilize cost-effective prompting solutions that offer a good balance between cost and performance.
- **Redundant Data Elimination**: Identify and eliminate redundant data to reduce storage requirements and associated costs.

5. **Add SQL Document Database to the RAG System's Knowledge:**
    - Integrate the SQL Document Database into the existing RAG System to enhance its knowledge base.
    - Ensure that the data is properly indexed and accessible for efficient querying.
    - Develop a comprehensive schema that supports the diverse range of documents and data types expected.
    - Test the Natural Language to SQL Langchain integration thoroughly to confirm that the database performs well under various load conditions.

By following these steps, the development of the financial RAG system will be enhanced, leading to a more efficient, cost-effective, and user-friendly application.