Financial Statement Analysis Using RAG Pipeline

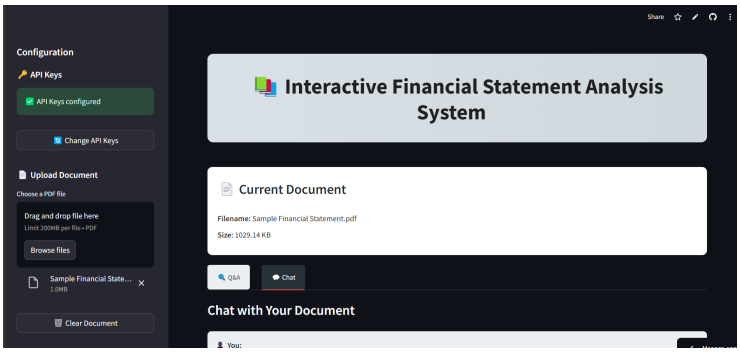

Figure 1: Main page image

Introduction

Retrieval-Augmented Generation (RAG) pipelines combine information retrieval and generative AI models to answer complex queries by extracting relevant context from large datasets and generating coherent responses. This document explains the architecture, data preprocessing, response generation, and challenges encountered while analyzing financial statements.

Model Architecture

The architecture of the RAG pipeline is composed of the following components:

1. **Document Parsing:** *LlamaParse* is used to extract and preprocess data from financial statements in PDF format, converting them into Markdown for easier processing.

2. **Node Processing:** Documents are segmented into manageable nodes using a *SimpleNodeParser*. This ensures efficient indexing and retrieval.

3. **Vector Store Indexing:** A *ChromaVectorStore* is employed to store the embeddings of document nodes for similarity-based retrieval.

4. **Query Engine:** The RAG pipeline integrates a query engine that retrieves relevant nodes from the vector store and generates responses using the *Gemini LLM*.

Approach to Data Extraction and Preprocessing

PDF Parsing

Financial statements often include complex data, such as tables, images, and structured text. *LlamaParse* converts these PDFs into Markdown format, preserving tables, text, and structural elements. This transformation simplifies subsequent data extraction and enables accurate embedding generation.

Node Processing

The documents are processed into nodes using the *SimpleNodeParser* with specified chunk sizes (e.g., 1024 tokens) and overlaps (e.g., 200 tokens). This segmentation ensures:

- Adequate context for embeddings.
- Efficient retrieval and response generation.

Vector Store Creation

Using *ChromaVectorStore*, the embeddings generated by *GeminiEmbedding* are stored. The vector store employs a cosine similarity metric for efficient retrieval of related nodes.

Generative Response Creation

The query engine is configured to retrieve the top 3 most relevant nodes and generate concise responses. Key configurations include:

- **Model:** *Gemini-1.5-flash* for large language modeling.
- **Embedding:** *GeminiEmbedding* with 768 dimensions.
- **Response Mode:** Compact responses to ensure clarity.

Challenges and Solutions

Challenge 1: Limited Effectiveness of Simple Vector Stores

Simple vector-based RAG pipelines struggled with datasets containing tabular data. Textual embeddings alone failed to capture the structure and context of financial tables.

Solution: The *LlamaParse* tool was introduced to parse PDFs into Markdown format, preserving the tabular structure and allowing seamless integration into the pipeline. This tool effectively captures complex data and converts it into a format suitable for processing.

Challenge 2: Maintaining Context During Parsing

PDF parsing sometimes resulted in fragmented or misaligned content, particularly with complex layouts in financial statements.

Solution: *LlamaParse* was configured with a content guideline instruction to extract all text and structural elements accurately, ensuring that no critical data was lost during the parsing process.

Challenge 3: Scaling the RAG Pipeline

Large datasets with many nodes posed indexing and retrieval challenges, particularly with the increasing size and complexity of financial documents.

Solution: The pipeline employed *ChromaVectorStore* with optimized storage and retrieval techniques, ensuring scalable and efficient operations for large datasets.

Step-by-Step Architecture

1. Document Parsing

- Tool: *LlamaParse*
- Output: Markdown with preserved tables, images, and structured text.

2. Node Processing

- Tool: *SimpleNodeParser*
- Parameters: Chunk size = 1024 tokens, Chunk overlap = 200 tokens.

3. Vector Store Indexing

- Tool: *ChromaVectorStore*
- Metric: Cosine similarity.

4. Query Engine and Response Generation

- LLM: *Gemini-1.5-flash*
- Embeddings: *GeminiEmbedding*
- Query Parameters: Top 3 similar nodes, compact responses.

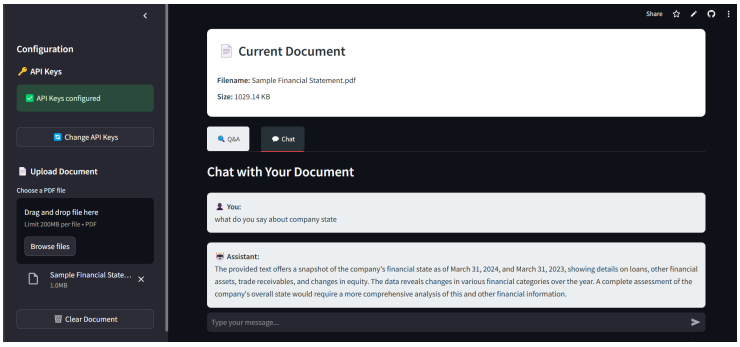

Figure 2: Chat bot chat image

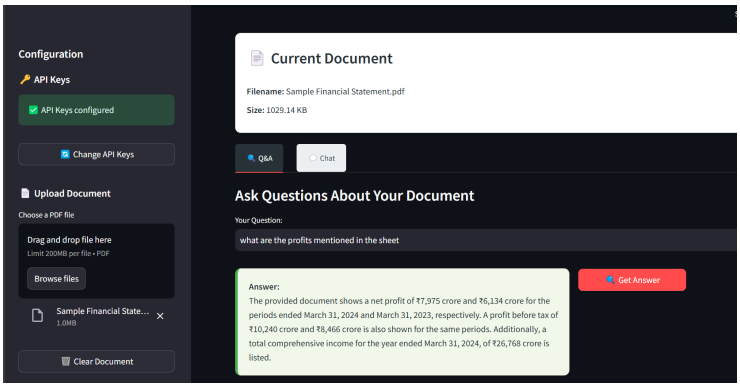

Figure 3: Q/A section image

Code Implementation

The code implementation includes the steps of setting up the pipeline, parsing documents, processing nodes, creating a vector store, and running the query engine.

```
# -*- coding: utf-8 -*-
"""Financial Statement Analysis using RAG Pipeline"""
```

```
import nest_asyncio
import chromadb
import os
from llama_index.llms.gemini import Gemini
from llama_index.core import Settings, StorageContext, VectorStoreIndex
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.embeddings.gemini import GeminiEmbedding
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_parse import LlamaParse
import google.generativeai as genai

nest_asyncio.apply()

class DocumentProcessor:
    def __init__(self, gemini_api_key: str, llama_cloud_api_key: str):
        self.gemini_api_key = gemini_api_key
        self.llama_cloud_api_key = llama_cloud_api_key
        genai.configure(api_key=self.gemini_api_key)
        self.llm = Gemini(model="models/gemini-1.5-flash", api_key=self.gemini_a
        self.embed_model = GeminiEmbedding(model="models/embedding-001", api_key
        Settings.llm = self.llm
        Settings.embed_model = self.embed_model
        self.chroma_client = chromadb.PersistentClient(path="./chroma_db")

    def parse_document(self, file_path: str):
        parser = LlamaParse(api_key=self.llama_cloud_api_key, result_type="markd
        documents = parser.load_data(file_path)
        return documents

    def process_nodes(self, documents):
        node_parser = SimpleNodeParser.from_defaults(chunk_size=1024, chunk_over
        nodes = node_parser.get_nodes_from_documents(documents)
        return nodes

    def create_vector_store(self, collection_name: str):
        collection = self.chroma_client.get_collection(name=collection_name)
        vector_store = ChromaVectorStore(chroma_collection=collection)
        return vector_store

    def create_rag_pipeline(self, nodes, collection_name):
        vector_store = self.create_vector_store(collection_name)
        storage_context = StorageContext.from_defaults(vector_store=vector_store
        index = VectorStoreIndex(nodes=nodes, storage_context=storage_context)
        query_engine = index.as_query_engine(similarity_top_k=3, response_mode="
        return query_engine
```

```
class QueryProcessor:
    def __init__(self, query_engine):
        self.query_engine = query_engine

    def process_query(self, query: str):
        response = self.query_engine.query(query)
        return str(response)

def main():
    gemini_api_key = "your_gemini_api_key"
    llama_cloud_api_key = "your_llama_cloud_api_key"
    processor = DocumentProcessor(gemini_api_key, llama_cloud_api_key)
    pdf_path = "/path/to/Sample Financial Statement.pdf"
    documents = processor.parse_document(pdf_path)
    nodes = processor.process_nodes(documents)
    query_engine = processor.create_rag_pipeline(nodes, "financial_statement_col
    query_processor = QueryProcessor(query_engine)

    question = "How do the net income and operating expenses compare for Q1 2024
    response = query_processor.process_query(question)
    print(f"Answer: {response}")

if __name__ == "__main__":
    main()
```

Conclusion

The RAG pipeline for financial statement analysis successfully integrates PDF parsing, vector indexing, and generative AI to provide insightful responses. Challenges with tabular data and scaling were addressed through thoughtful tool selection and configuration, ensuring robust performance for complex financial datasets.