Intelligent Document Chunking and RAG with Groq LLM

This repository implements an advanced Retrieval-Augmented Generation (RAG) system using various document chunking strategies, embedding models, and integrates with Groq's LLM API for improved query understanding and response generation.

🌟 Project Overview

This project demonstrates different approaches to document chunking for structured data, vector embeddings, and integration with Large Language Models (LLMs) to create an effective information retrieval and question answering system. The core components include:

- **Multiple Chunking Strategies**: Semantic, Fixed, Recursive, Documentation, and Agentic chunking
- **Embedding Generation**: Using Sentence Transformers to create vector representations
- **Vector Database Storage**: FAISS for efficient similarity search
- **Query Refinement**: Using Groq LLM to refine user queries for better retrieval
- **Answer Generation**: RAG architecture for context-aware responses

🔧 Technical Architecture

User Query → Query Refinement (Groq) → Vector Search (FAISS) → Context Retrieval → Answer Generation (Groq)

📋 Features

- **Semantic Chunking**: Chunks text based on semantic meaning
- **Fixed Chunking**: Divides text into uniform-sized chunks
- **Recursive Chunking**: Splits text recursively with configurable parameters
- **Documentation Chunking**: Intelligent chunking based on paragraph breaks
- **Agentic Chunking**: Advanced chunking using hybrid paragraph and size constraints
- **Query Enhancement**: Uses LLM to refine user queries for improved accuracy
- **Context-Aware Response Generation**: Leverages retrieved context for accurate answers

🛠️ Installation

Clone the repository

git clone https://github.com/yourusername/chunking-rag-groq.git

cd chunking-rag-groq

Install dependencies

pip install pandas numpy sentence-transformers faiss-cpu scikit-learn groq

pip install langchain langchain-community PyPDF2 pymupdf transformers

🔑 Configuration

Before running the code, you need to:

1. Configure your Groq API key:
2. client = Groq(api_key="your_groq_api_key")
3. Prepare your data in Excel format with at least two columns:
    - Problem: The text data you want to chunk and query
    - Solution: Optional reference solutions

🚀 Usage

To run each chunking method, execute the corresponding Python script:

Example for Semantic Chunking

python semantic_chunking.py

When prompted, enter your query

Enter your query or problem: Your problem statement here

Each script will:

1. Load and preprocess the Excel data
2. Apply the specific chunking strategy
3. Create embeddings and build a vector store
4. Refine your query using Groq LLM
5. Retrieve relevant context from the vector store
6. Generate a comprehensive answer using Groq LLM with the retrieved context

💡 Chunking Strategies Explained

Semantic Chunking

Splits text based on semantic meaning while maintaining context.

```
splitter = CharacterTextSplitter(chunk_size=100, chunk_overlap=10)
```

Fixed Chunking

Creates uniform chunks of fixed size without semantic consideration.

```
splitter = CharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=0)
```

Recursive Chunking

Recursively splits text with varying chunk sizes based on content structure.

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=min(max_chunk_size, 1000),
    chunk_overlap=20,
    length_function=len
)
```

Documentation Chunking

Chunks text based on natural paragraph breaks in documentation.

```
# Custom function to split by paragraph breaks
def documentation_chunking(documents):
    chunks = []
    for doc in documents:
        paragraphs = doc.page_content.split('\n\n')
        for paragraph in paragraphs:
            if paragraph.strip():
                chunks.append(paragraph.strip())
    return chunks
```

Agentic Chunking

Hybrid approach that considers both paragraph structure and size constraints.

```
def agentic_chunking(documents, chunk_size=300):
    chunks = []
    current_chunk = ""

    for doc in documents:
        for paragraph in doc.page_content.split('\n\n'):
            paragraph = paragraph.strip()
            if not paragraph:
                continue

            if len(current_chunk) + len(paragraph) + 1 <= chunk_size:
                current_chunk += f"{paragraph}\n\n"
            else:
                if current_chunk:
                    chunks.append(current_chunk.strip())
                current_chunk = f"{paragraph}\n\n"

    if current_chunk:
        chunks.append(current_chunk.strip())

    return chunks
```

📊 **Comparative Analysis**

Each chunking strategy offers different advantages:

Strategy	Best For	Advantages	Limitations
Semantic	Natural language text	Preserves meaning	Requires tuning
Fixed	Simple texts	Predictable size	May split coherent ideas
Recursive	Hierarchical content	Respects structure	More complex to implement
Documentation	Technical docs	Preserves paragraph integrity	Depends on formatting
Agentic	Mixed content	Balances size and meaning	Most compute-intensive

🔍 Query Refinement Process

The system uses Groq's LLM to refine user queries for improved retrieval:

```
def refine_query_with_groq(query_text):
    chat_completion = client.chat.completions.create(
        messages=[
            {
                "role": "system",
                "content": "You are an AI assistant that refines user queries to improve search accuracy. Provide a concise, refined version of the user's query.",
            },
            {
                "role": "user",
                "content": f"Refine the following problem description for better search accuracy: {query_text}",
            }
        ],
        model="llama3-8b-8192",
        max_tokens=100,
```

```
    )
    return chat_completion.choices[0].message.content.strip()
```

📝 RAG Implementation

The RAG (Retrieval-Augmented Generation) architecture combines retrieved context with LLM generation:

```
def generate_answer_with_rag(refined_query_text, retrieval_context):
    chat_completion = client.chat.completions.create(
        messages=[
            {
                "role": "system",
                "content": "You are an AI assistant that generates answers to user queries based on retrieved context.",
            },
            {
                "role": "user",
                "content": f"Using the following retrieved context, provide a detailed solution to the query: {retrieval_context}\nQuery: {refined_query_text}",
            }
        ],
        model="llama3-8b-8192",
        max_tokens=500,
    )
    return chat_completion.choices[0].message.content.strip()
```

🧪 Future Improvements

- Implement hybrid search combining semantic and keyword-based approaches
- Add prompt templates for more sophisticated prompt engineering
- Incorporate feedback loops to improve retrieval quality

- Add support for additional file formats (PDF, CSV, etc.)
- Implement evaluation metrics to compare chunking strategies

🙏 Acknowledgements

- LangChain for the document processing framework
- Sentence Transformers for embedding models
- FAISS for efficient similarity search
- Groq for their LLM API

👥 Contact

For questions or suggestions, please open an issue in this repository or contact [jathingangi@gmail.com].