Jahnavi Pakanati-9013742

Team-6

Sustainable-AI

What Is RAG: - RAG stands for Retrieval-Augmented Generation. It's an AI approach that makes large language models (LLMs) smarter, more accurate, and more reliable by letting them look things up before answering.

What It does:

RAG retrieves relevant information from a knowledge base and includes it in the prompt given to the LLM.

How is it relevant with this project?

Here we were initially doing a search on large document base but then later we dropped the idea of RAG and we are not using it.

How RAG works:

```python
# Test function
def test_persistent_medical_rag_system():
    """Test the RAG system with persistent medical embeddings"""
    print("🔬 Testing Persistent Medical RAG System with MedEmbed...")

    try:
        # Initialize with persistent medical embeddings
        rag = RAGSystem(embedding_model="medical")

        # Test persistence verification
        persistence_ok = rag.verify_persistence()
        print(f"📊 Persistence verification: {'✅ PASSED' if persistence_ok else '❌ FAILED'}")

        # Test system stats
        stats = rag.get_system_stats()
        print(f"📊 Persistent medical system stats: {stats}")

        # Test medical search
        medical_queries = [
            "chest pain and shortness of breath",
            "diabetes type 2 management",
            "post operative complications"
        ]

        for query in medical_queries:
            results = rag.search_relevant_context(query)
            print(f"🔍 '{query}': {len(results)} medical documents found (persistent)")

        print("✅ Persistent Medical RAG System test completed successfully")
        return True

    except Exception as e:
        print(f"❌ Persistent Medical RAG System test failed: {e}")
        traceback.print_exc()
        return False
```

1. **Retrieval Phase:**
   Function used : search_relevant_context(query, max_docs=5)

- **Process:**
  Prints a start message indicating the test has begun.
- Creates a RAGSystem instance using the medical embedding model (MedEmbed).
- Verifies persistent ChromaDB storage and prints ✅ PASSED or ❌ FAILED.
- Retrieves and prints system stats (doc count, storage type, model, GPU availability, etc.).
- Defines three sample medical queries for testing.
- Runs a search for each query, retrieving relevant document chunks and printing how many were found.
- If all steps succeed, prints a success message and returns True.
- If any error occurs, prints an error message, shows traceback details, and returns False.

```python
def verify_persistence(self) -> bool:
    """
    ✅ NEW: Verify that documents will persist across application restarts
    """
    if not self.chroma_collection:
        print("❌ No persistent collection available")
        return False

    try:
        # Check if ChromaDB files exist on disk
        chroma_files = [
            "chroma.sqlite3",  # Main database file
            "index",           # Index files directory
        ]

        persistence_files_found = []
        for file_name in chroma_files:
            file_path = os.path.join(self.data_dir, file_name)
            if os.path.exists(file_path):
                persistence_files_found.append(file_name)

        if persistence_files_found:
            print(f"✅ Persistence files found: {persistence_files_found}")
            doc_count = self.chroma_collection.count()
            print(f"✅ Persistent storage verified: {doc_count} documents will persist")
            return True
        else:
            print("❌ No persistence files found on disk")
            return False

    except Exception as e:
        print(f"❌ Persistence verification failed: {e}")
        return False
```

- Checks if a persistent ChromaDB collection exists; if not, returns False.
- Defines the expected persistence files: chroma.sqlite3 (database) and index (vector index).
- Scans the storage directory to see if these files exist.
- If found, prints the file names and document count, then returns True.
- If not found, prints a warning and returns False.
- Catches and reports any errors during the process.

```python
def get_all_documents(self) -> List[Dict[str, Any]]:
    """Get all document metadata"""
    try:
        with sqlite3.connect(self.db_path) as conn:
            cursor = conn.execute('SELECT * FROM documents ORDER BY processed_at DESC')
            columns = [desc[0] for desc in cursor.description]

            docs = []
            for row in cursor.fetchall():
                doc = dict(zip(columns, row))
                docs.append(doc)

            return docs
    except Exception as e:
        logging.error(f"Failed to get documents: {e}")
        return []
```

- It connects to the database file (self.db_path).
- Runs a query to get every row from the documents table, sorted so the most recently processed ones come first.
- It converts each row into a dictionary where the keys are column names and the values are the corresponding data.
- All these dictionaries are collected into a list and returned.
- If anything goes wrong (e.g., database file missing, query error), it logs the problem and returns an empty list instead.

Purpose

- This method stores document metadata into a SQLite database table called documents.
- Used for keeping a record of processed documents for tracking and retrieval.

Generate Unique ID

- Uses uuid.uuid4() to create a unique document ID (doc_id).
- Ensures no two documents have the same identifier.

Database Connection

- Connects to the SQLite database located at self.db_path using sqlite3.connect().
- The with statement ensures the connection is automatically closed after use.

Insert or Replace Data

- Executes an INSERT OR REPLACE SQL query:
- Inserts a new row if no record exists with this id.
- Replaces (updates) the row if the same id already exists.

Data Stored in Table

- id → Generated doc_id.
- file_name → Taken from metadata, defaults to empty string.
- doc_type → Defaults to "medical" if not in metadata.
- language → Defaults to "en".
- file_size → Defaults to 0.
- processed_at → Defaults to empty string.

- embedding_model → Defaults to empty string.
- metadata → The entire metadata dictionary converted into a string.
- Safe Parameter Binding
- Uses placeholders in SQL to avoid SQL injection attacks.
- Automatic Resource Management
- The with block commits the transaction automatically and closes the connection.
- Error Handling
- If any exception occurs (e.g., missing table, DB file issues), it is caught and logged using logging.error().