Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Subject: NLP Branch: AIML

Name	Jash Vora
UID	2022600065
Batch	A1
Course	NLP
Exp no.	8
Name of the Experiment	Retrieval Augmented Generation (RAG) based Gen-AI Tool
AIM	Build a Gen-AI tool using a vector database. Students will select a topic from the provided list (image).
Theory	Retrieval Augmented Generation (RAG): Combines information retrieval with a generative model. ● Workflow: ○ Convert documents into embeddings (vector representation). ○ Store them in a vector database. ○ On a query, retrieve top-k relevant chunks. ○ Pass retrieved context to an LLM to generate an accurate, grounded answer. ● Advantages: ○ Uses latest data without re-training model.

○ Reduces hallucination by grounding responses in retrieved knowledge.

Code

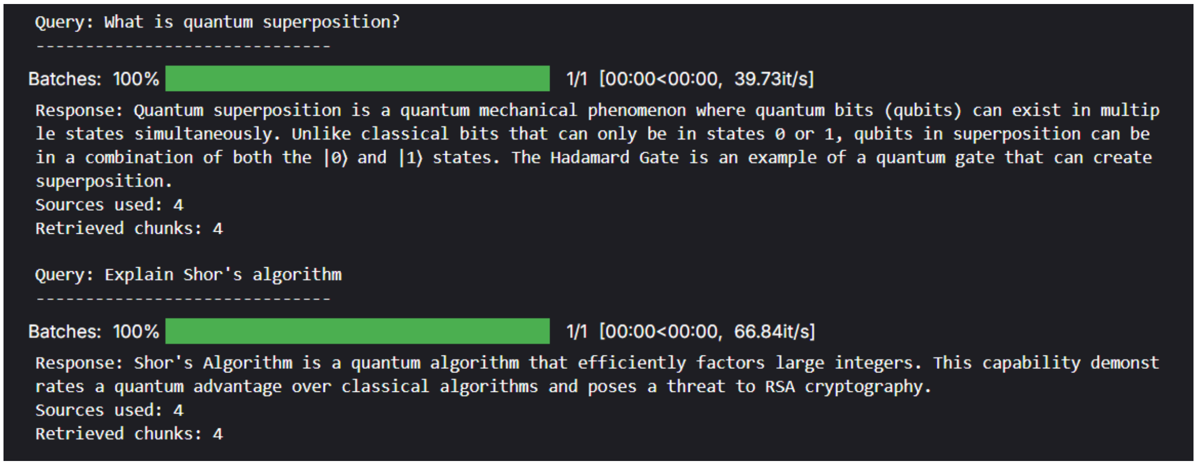
```python
import os
import numpy as np
import google.generativeai as genai
from typing import List, Dict, Any, Optional
import requests
from bs4 import BeautifulSoup
import PyPDF2
import io
from sentence_transformers import SentenceTransformer
import json
import pickle
from dataclasses import dataclass
from datetime import datetime
import logging
import faiss

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class Document:
    """Represents a document in the knowledge base"""
    content: str
    metadata: Dict[str, Any]
    id: str
    embedding: Optional[List[float]] = None

class FAISSVectorStore:
    """FAISS-based vector storage for the RAG system"""

    def __init__(self, dimension: int = 384, index_path: str = "./faiss_index"):
        self.dimension = dimension
        self.index_path = index_path
        self.index = faiss.IndexFlatIP(dimension)  # Inner product
similarity
        self.documents = {}  # Store document content and metadata
        self.id_to_index = {}  # Map document IDs to FAISS indices
        self.next_index = 0

        # Try to load existing index
        self.load_index()

    def add_documents(self, documents: List[str], embeddings:
List[List[float]],
                     metadatas: List[Dict], ids: List[str]):
        """Add documents to the FAISS index"""
        embeddings_array = np.array(embeddings, dtype=np.float32)

        # Normalize embeddings for cosine similarity
        faiss.normalize_L2(embeddings_array)

        # Add to FAISS index
        start_index = self.next_index
        self.index.add(embeddings_array)

        # Store documents and metadata
        for i, (doc, metadata, doc_id) in enumerate(zip(documents,
metadatas, ids)):
            current_index = start_index + i
            self.documents[current_index] = {
                'content': doc,
                'metadata': metadata,
                'id': doc_id
            }
            self.id_to_index[doc_id] = current_index

        self.next_index += len(documents)
        self.save_index()

    def search(self, query_embedding: List[float], k: int = 5) ->
List[Dict]:
        """Search for similar documents"""
        if self.index.ntotal == 0:
            return []

        query_array = np.array([query_embedding], dtype=np.float32)
        faiss.normalize_L2(query_array)

        # Search
        scores, indices = self.index.search(query_array, min(k,
self.index.ntotal))

        results = []
        for score, idx in zip(scores[0], indices[0]):
            if idx >= 0 and idx in self.documents:
                result = self.documents[idx].copy()
                result['score'] = float(score)
                results.append(result)

        return results

    def count(self) -> int:
        """Get total number of documents"""
        return self.index.ntotal

    def save_index(self):
        """Save FAISS index and metadata to disk"""
        try:
            os.makedirs(os.path.dirname(self.index_path),
exist_ok=True)

            # Save FAISS index
            faiss.write_index(self.index, f"{self.index_path}.faiss")

            # Save metadata
            metadata = {
                'documents': self.documents,
                'id_to_index': self.id_to_index,
                'next_index': self.next_index,
                'dimension': self.dimension
            }
            with open(f"{self.index_path}_metadata.pkl", 'wb') as f:
                pickle.dump(metadata, f)

        except Exception as e:
            logger.error(f"Error saving index: {e}")

    def load_index(self):
        """Load FAISS index and metadata from disk"""
        try:
            index_file = f"{self.index_path}.faiss"
            metadata_file = f"{self.index_path}_metadata.pkl"

            if os.path.exists(index_file) and os.path.exists(metadata_file):
                # Load FAISS index
                self.index = faiss.read_index(index_file)

                # Load metadata
                with open(metadata_file, 'rb') as f:
                    metadata = pickle.load(f)
                    self.documents = metadata.get('documents', {})
                    self.id_to_index = metadata.get('id_to_index', {})
                    self.next_index = metadata.get('next_index', 0)

                logger.info(f"Loaded existing index with {self.index.ntotal} documents")
            else:
                logger.info("No existing index found, starting fresh")

        except Exception as e:
            logger.error(f"Error loading index: {e}")
            logger.info("Starting with fresh index")


class QuantumRAGAgent:
    def __init__(self, gemini_api_key: str = None):
        """
        Initialize the Quantum Computing RAG Agent
        Args:
            gemini_api_key: Gemini API key (will use .env if not
provided)
        """
        # Load API key
        if gemini_api_key:
            self.api_key = gemini_api_key
        else:
            self.api_key = secret_value_0

        if not self.api_key:
            raise ValueError("Gemini API key not found. Please set
GEMINI_API_KEY in .env or pass it directly.")

        # Configure Gemini
        genai.configure(api_key=self.api_key)
        self.model = genai.GenerativeModel('gemini-2.5-flash')

        # Initialize embedding model
        self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')

        # Initialize FAISS vector store
        self.vector_store = FAISSVectorStore(dimension=384)  #
all-MiniLM-L6-v2 dimension

        logger.info("Quantum RAG Agent initialized successfully")

    def extract_text_from_pdf(self, pdf_path: str) -> str:
        """Extract text from PDF file"""
        text = ""
        try:
            with open(pdf_path, 'rb') as file:
                pdf_reader = PyPDF2.PdfReader(file)
                for page in pdf_reader.pages:
                    text += page.extract_text() + "\n"
        except Exception as e:
            logger.error(f"Error extracting text from PDF: {e}")
        return text
```

```python
    def extract_text_from_url(self, url: str) -> str:
        """Extract text from web URL"""
        try:
            response = requests.get(url, timeout=10)
            response.raise_for_status()
            soup = BeautifulSoup(response.content, 'html.parser')

            # Remove script and style elements
            for script in soup(["script", "style"]):
                script.decompose()

            # Get text and clean it
            text = soup.get_text()
            lines = (line.strip() for line in text.splitlines())
            chunks = (phrase.strip() for line in lines for phrase in
line.split("  "))
            text = ' '.join(chunk for chunk in chunks if chunk)

            return text
        except Exception as e:
            logger.error(f"Error extracting text from URL {url}: {e}")
            return ""

    def chunk_text(self, text: str, chunk_size: int = 1000, overlap:
int = 200) -> List[str]:
        """Split text into overlapping chunks"""
        words = text.split()
        chunks = []

        for i in range(0, len(words), chunk_size - overlap):
            chunk = ' '.join(words[i:i + chunk_size])
            chunks.append(chunk)

            if i + chunk_size >= len(words):
                break

        return chunks
    def add_document(self, content: str, metadata: Dict[str, Any],
doc_id: str = None) -> str:
        """Add a document to the knowledge base"""
        if not doc_id:
            doc_id = f"doc_{datetime.now().timestamp()}"

        # Chunk the document
        chunks = self.chunk_text(content)

        # Prepare data for FAISS
        documents = []
        embeddings = []
        metadatas = []
        ids = []

        # Generate embeddings and prepare data
        for i, chunk in enumerate(chunks):
            chunk_id = f"{doc_id}_chunk_{i}"
            embedding = self.embedding_model.encode(chunk).tolist()

            documents.append(chunk)
            embeddings.append(embedding)
                         metadatas.append({**metadata, "chunk_id": i,
"parent_doc_id": doc_id})
            ids.append(chunk_id)

        # Add to FAISS vector store
                self.vector_store.add_documents(documents, embeddings,
metadatas, ids)

             logger.info(f"Added document {doc_id} with {len(chunks)}
chunks")
        return doc_id

    def add_pdf(self, pdf_path: str, metadata: Dict[str, Any] = None)
-> str:
        """Add a PDF document to the knowledge base"""
        if not metadata:
            metadata = {"source": pdf_path, "type": "pdf"}
        text = self.extract_text_from_pdf(pdf_path)
        if not text.strip():
            logger.warning(f"No text extracted from PDF: {pdf_path}")
            return None

        return self.add_document(text, metadata)

    def add_url(self, url: str, metadata: Dict[str, Any] = None) -> str:
        """Add content from a URL to the knowledge base"""
        if not metadata:
            metadata = {"source": url, "type": "web"}

        text = self.extract_text_from_url(url)
        if not text.strip():
            logger.warning(f"No text extracted from URL: {url}")
            return None

        return self.add_document(text, metadata)

    def add_text(self, text: str, metadata: Dict[str, Any] = None) -> str:
        """Add raw text to the knowledge base"""
        if not metadata:
            metadata = {"type": "text", "added_at": datetime.now().isoformat()}

        return self.add_document(text, metadata)

    def initialize_quantum_knowledge_base(self):
        """Initialize with basic quantum computing knowledge"""
        quantum_topics = [
            {
                "content": """
                Quantum Computing Fundamentals:
                    Quantum computing is a revolutionary computational paradigm that leverages quantum mechanical phenomena
```

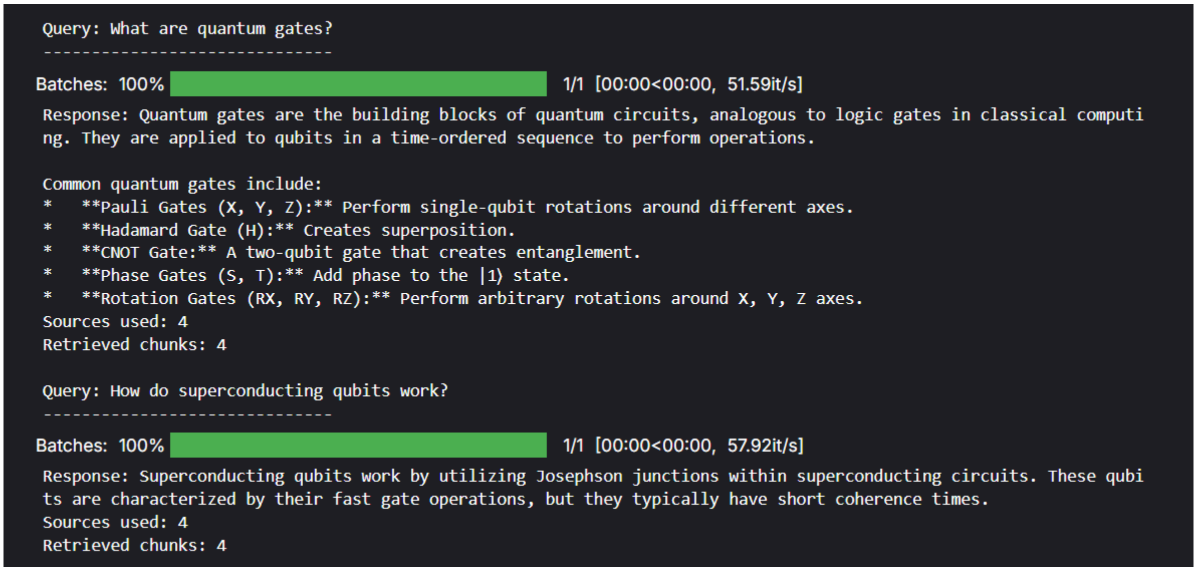
```
                  such as superposition, entanglement, and interference
to process information. Unlike classical bits
                      that can only be in states 0 or 1, quantum bits
(qubits) can exist in superposition states,
                allowing them to be in multiple states simultaneously.

                Key principles:
                   1. Superposition: Qubits can be in a combination of
|0⟩ and |1⟩ states
                  2. Entanglement: Quantum particles can be correlated
in ways that classical physics cannot explain
                     3. Interference: Quantum amplitudes can interfere
constructively or destructively
                  4. Measurement: Observing a quantum system collapses
it to a definite state
                """,
                        "metadata": {"topic": "fundamentals", "type":
"theory"}
            },
            {
                "content": """
                Quantum Gates and Circuits:
                     Quantum gates are the building blocks of quantum
circuits, analogous to logic gates in classical computing.
                Common quantum gates include:

                     1. Pauli Gates (X, Y, Z): Single-qubit rotations
around different axes
                       2. Hadamard Gate (H): Creates superposition by
rotating |0⟩ to (|0⟩ + |1⟩)/√2
                3. CNOT Gate: Two-qubit gate that creates entanglement
                4. Phase Gates (S, T): Add phase to |1⟩ state
                   5. Rotation Gates (RX, RY, RZ): Arbitrary rotations
around X, Y, Z axes

                     Quantum circuits are represented as sequences of
quantum gates applied to qubits,
                read from left to right in time order.
                """,
                "metadata": {"topic": "gates_circuits", "type": "theory"}
            },
            {
                "content": """
                Quantum Algorithms:
                Several quantum algorithms demonstrate quantum advantage over classical algorithms:

                1. Shor's Algorithm: Efficiently factors large integers, threatening RSA cryptography
                2. Grover's Algorithm: Searches unsorted databases with quadratic speedup
                3. Quantum Fourier Transform: Essential subroutine for many quantum algorithms
                4. Variational Quantum Eigensolver (VQE): Hybrid quantum-classical algorithm for finding ground states
                5. Quantum Approximate Optimization Algorithm (QAOA): For solving combinatorial optimization problems
                6. Deutsch-Jozsa Algorithm: Determines if a function is constant or balanced

                These algorithms exploit quantum phenomena to achieve computational advantages
                in specific problem domains.
                """,
                "metadata": {"topic": "algorithms", "type": "theory"}
            },
            {
                "content": """
                Quantum Hardware Platforms:
                Different physical systems are used to implement quantum computers:

                1. Superconducting Qubits: Used by IBM, Google, Rigetti
                   - Josephson junctions in superconducting circuits
                   - Fast gate operations, but short coherence times
                2. Trapped Ion Systems: Used by IonQ, Honeywell
                   - Individual ions trapped by electromagnetic fields
                     - High fidelity, long coherence times, but slower
gates

                3. Photonic Systems: Used by Xanadu, PsiQuantum
                   - Photons as qubits, room temperature operation
                   - Natural for quantum communication

                4. Neutral Atoms: Used by QuEra, Pasqal
                   - Atoms trapped by optical tweezers
                   - Highly scalable architectures

                5. Silicon Spin Qubits: Emerging technology
                   - Compatible with semiconductor manufacturing
                """,
                "metadata": {"topic": "hardware", "type":
"technology"}
            }
        ]

        for topic in quantum_topics:
            self.add_text(topic["content"], topic["metadata"])

        logger.info("Initialized quantum computing knowledge base")

    def retrieve_relevant_docs(self, query: str, n_results: int = 5)
-> List[Dict[str, Any]]:
        """Retrieve relevant documents for a query"""
        # Generate query embedding
        query_embedding = self.embedding_model.encode(query).tolist()

        # Search in FAISS vector store
        results = self.vector_store.search(query_embedding, n_results)

        return results

    def generate_response(self, query: str, context_docs:
List[Dict[str, Any]]) -> str:
        """Generate response using Gemini with retrieved context"""
        # Prepare context from retrieved documents
        context = "\n\n".join([
                  f"[Source: {doc['metadata'].get('source', 'Knowledge
Base')}]\n{doc['content']}"
            for doc in context_docs
        ])

        # Shortened prompt to avoid token limits
         prompt = f"""Based on the quantum computing context provided,
answer this question: {query}

Context:
{context}

Provide a clear, accurate answer using the context above. Focus on the
key concepts and be concise."""

        try:
            # Add generation config to handle potential issues
            generation_config = genai.types.GenerationConfig(
                candidate_count=1,
                max_output_tokens=1000,  # Limit output tokens
                temperature=0.3,
            )

            response = self.model.generate_content(
                prompt,
                generation_config=generation_config
            )

            # Check if response was blocked
                                        if   response.candidates   and
response.candidates[0].content.parts:
                return response.text
            else:
                logger.warning("Response was blocked or empty")
                   return "I couldn't generate a response. The content
may have been filtered. Please try rephrasing your question."
```

```
        except Exception as e:
            logger.error(f"Error generating response: {e}")
            # More specific error handling
            if "quota" in str(e).lower() or "limit" in str(e).lower():
                return "API quota exceeded. Please check your Gemini API usage limits."
            elif "safety" in str(e).lower() or "blocked" in str(e).lower():
                return "Response was blocked by safety filters. Please try rephrasing your question."
            else:
                return f"Error generating response: {str(e)}"

    def query(self, question: str, n_results: int = 5) -> Dict[str, Any]:
        """Main query method - retrieve and generate response"""
        logger.info(f"Processing query: {question}")

        # Retrieve relevant documents
        relevant_docs = self.retrieve_relevant_docs(question, n_results)

        if not relevant_docs:
            return {
                "query": question,
                "response": "No relevant documents found in the knowledge base.",
                "sources": [],
                "retrieved_docs": 0
            }

        # Generate response
        response = self.generate_response(question, relevant_docs)

        return {
            "query": question,
            "response": response,
            "sources": [doc['metadata'] for doc in relevant_docs],
            "retrieved_docs": len(relevant_docs)
        }

    def get_collection_stats(self) -> Dict[str, Any]:
        """Get statistics about the knowledge base"""
        count = self.vector_store.count()
        return {
            "total_chunks": count,
            "backend": "FAISS"
        }

# Example usage and testing
def main():
    """Example usage of the Quantum RAG Agent"""

    try:
        # Initialize the agent
        agent = QuantumRAGAgent()

        # Initialize with basic quantum knowledge
        agent.initialize_quantum_knowledge_base()

        # Example queries
        example_queries = [
            "What is quantum superposition?",
            "Explain Shor's algorithm",
            "What are quantum gates?",
            "How do superconducting qubits work?"
        ]

        print("Quantum Computing RAG Agent - Example Queries\n")
        print("=" * 50)

        for query in example_queries[:4]:
            print(f"\nQuery: {query}")
            print("-" * 30)

            result = agent.query(query)
            print(f"Response: {result['response']}")
            print(f"Sources used: {len(result['sources'])}")
            print(f"Retrieved chunks: {result['retrieved_docs']}")

        # Print collection stats
        stats = agent.get_collection_stats()
        print(f"\nKnowledge Base Stats:")
        print(f"Total chunks: {stats['total_chunks']}")
        print(f"Backend: {stats['backend']}")

    except Exception as e:
        print(f"Error in main: {e}")
        print("Make sure you have set your GEMINI_API_KEY in a .env file or environment variable")

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

Output



```
Query: What is quantum superposition?
------------------------------
Batches: 100%|██████████| 1/1 [00:00<00:00, 39.73it/s]
Response: Quantum superposition is a quantum mechanical phenomenon where quantum bits (qubits) can exist in multip
le states simultaneously. Unlike classical bits that can only be in states 0 or 1, qubits in superposition can be
in a combination of both the |0⟩ and |1⟩ states. The Hadamard Gate is an example of a quantum gate that can create
superposition.
Sources used: 4
Retrieved chunks: 4

Query: Explain Shor's algorithm
------------------------------
Batches: 100%|██████████| 1/1 [00:00<00:00, 66.84it/s]
Response: Shor's Algorithm is a quantum algorithm that efficiently factors large integers. This capability demonst
rates a quantum advantage over classical algorithms and poses a threat to RSA cryptography.
Sources used: 4
Retrieved chunks: 4
```

<table>
<tr>
<td></td>
<td>


```
Query: What are quantum gates?
------------------------------
Batches: 100% ████████████████████ 1/1 [00:00<00:00, 51.59it/s]
Response: Quantum gates are the building blocks of quantum circuits, analogous to logic gates in classical computi
ng. They are applied to qubits in a time-ordered sequence to perform operations.

Common quantum gates include:
*   **Pauli Gates (X, Y, Z):** Perform single-qubit rotations around different axes.
*   **Hadamard Gate (H):** Creates superposition.
*   **CNOT Gate:** A two-qubit gate that creates entanglement.
*   **Phase Gates (S, T):** Add phase to the |1⟩ state.
*   **Rotation Gates (RX, RY, RZ):** Perform arbitrary rotations around X, Y, Z axes.
Sources used: 4
Retrieved chunks: 4

Query: How do superconducting qubits work?
------------------------------
Batches: 100% ████████████████████ 1/1 [00:00<00:00, 57.92it/s]
Response: Superconducting qubits work by utilizing Josephson junctions within superconducting circuits. These qubi
ts are characterized by their fast gate operations, but they typically have short coherence times.
Sources used: 4
Retrieved chunks: 4
```
</td>
</tr>
<tr>
<td>Conclusion</td>
<td>This experiment demonstrates how Retrieval Augmented Generation (RAG) can be used for specific tasks</td>
</tr>
</table>