Homework 7: Build a RAG System

Points: 20 | **Due:** Sunday, March 29, 2026 @ 11pm Pacific

Author: Richard Young, Ph.D. | UNLV Lee Business School

Compute: CPU (free tier) — GPU recommended

Learning Objectives

1. **Understand** what RAG is and why it matters
2. **Build** a document retrieval system using embeddings
3. **Integrate** retrieval with a language model
4. **Evaluate** RAG output quality
5. **Identify** when RAG helps vs. hurts

Why This Matters for Business

Enterprise Knowledge: Microsoft's Copilot uses RAG to answer questions about your company's documents. Without RAG, the LLM would hallucinate answers. With RAG, it quotes your actual policies, contracts, and reports.

Customer Support: Intercom reduced support costs by 67% with a RAG-powered chatbot that retrieves relevant help articles before generating responses—ensuring accurate answers grounded in documentation.

Legal Research: Harvey AI helps lawyers search millions of case documents. Pure LLMs would make up case citations; RAG ensures every reference comes from real legal documents.

Healthcare: Nuance's clinical assistants use RAG to retrieve relevant medical literature when helping doctors. Hallucinated medical advice could be dangerous; RAG keeps responses grounded in evidence.

Grading

Component	Points	Effort	What We're Looking For
Document Chunking	3	*	Split documents into retrievable chunks
Embedding Index	4	*	Create searchable vector store
Retrieval	5	**	Find relevant chunks for queries
Generation	5	**	Use retrieved context in LLM prompt
Evaluation	3	**	Assess quality, identify limitations
Total	**20**		

Effort Key: * Straightforward | ** Requires thinking | *** Challenge

---

The Big Picture

The Problem: LLMs have fixed knowledge (training cutoff) and can hallucinate.

RAG Solution: 1. **Retrieve:** Find relevant documents for the user's question 2. **Augment:** Add those documents to the LLM's prompt 3. **Generate:** LLM answers using the provided context

```
User Query → Embed → Search Vector DB → Retrieve Top-K Docs
                                                   ↓
                            LLM ← Augmented Prompt ← Combine Query + Docs
                             ↓
                       Grounded Answer
```

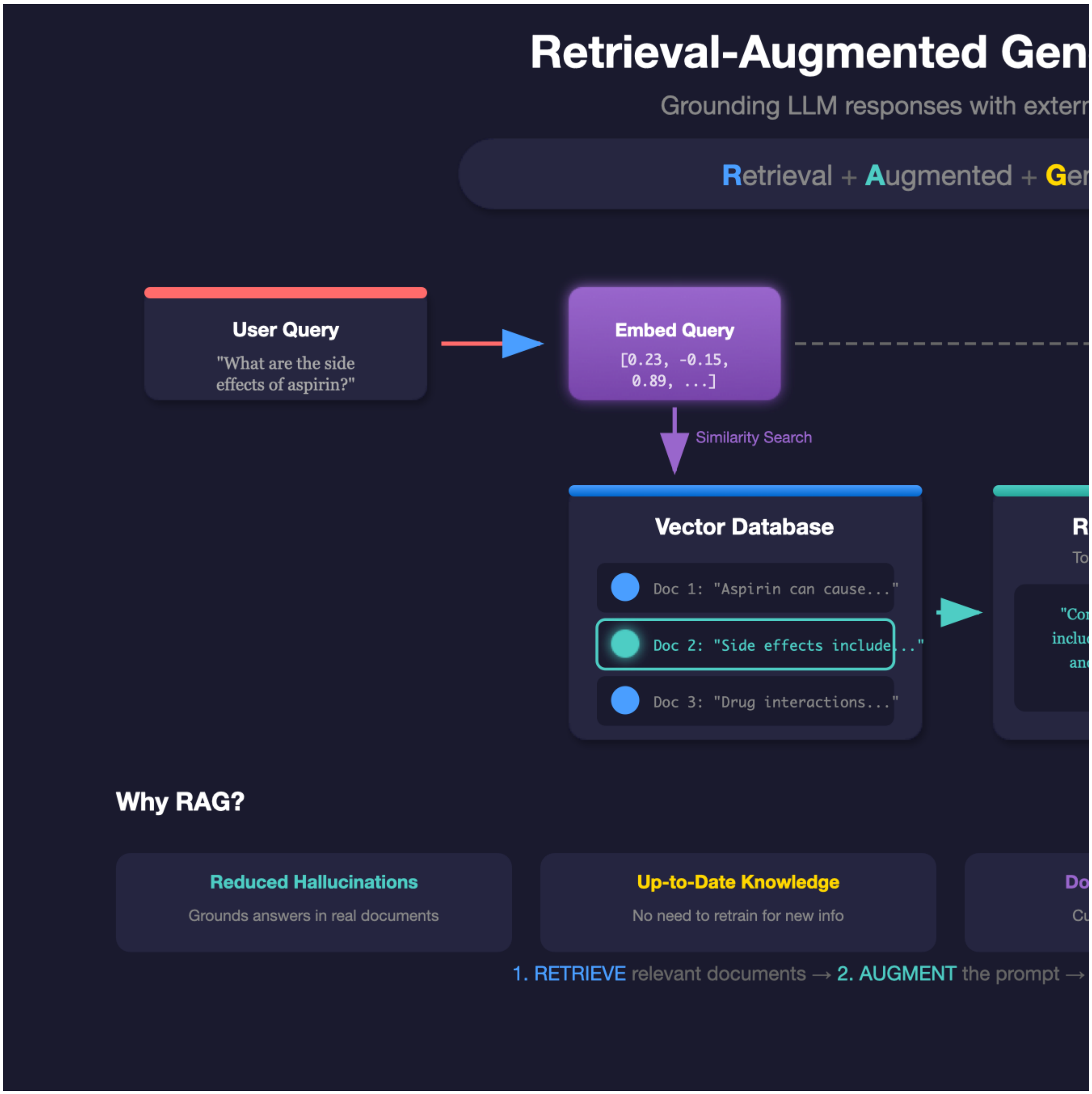

Figure 1: RAG Architecture

Instructions

1. Open `MIS769_HW7_RAG_System.ipynb` in Google Colab
2. Load your document corpus (reviews, articles, or provided dataset)
3. Chunk documents into retrievable segments
4. Create embeddings and build a vector index
5. Implement retrieval: given a query, find relevant chunks
6. Build the full RAG pipeline with an LLM
7. Test with various questions and evaluate quality

What Your Output Should Look Like

Document Chunking:

```
□  DOCUMENT CHUNKING
============================================================
Original documents: 5,000
Total chunks created: 23,456
Average chunk size: 312 tokens
Overlap: 50 tokens
```

Vector Index:

```
□  VECTOR INDEX BUILT
============================================================
Embeddings shape: (23456, 384)
Index type: FAISS (Flat L2)
Index size: 34.2 MB
```

Retrieval Test:

```
□  RETRIEVAL TEST
============================================================
Query: "What do customers say about battery life?"

Retrieved Chunks:
1. [0.89] "The battery lasts all day, even with heavy use..."
2. [0.85] "Disappointed with battery - only 4 hours..."
3. [0.82] "Charging takes forever but battery life is decent..."
```

RAG Response:

```
□  RAG RESPONSE
============================================================
Query: "What do customers say about battery life?"

Retrieved Context: [3 chunks shown above]

Generated Answer:
"Customer opinions on battery life are mixed. Some users report
```

```
excellent all-day battery life even with heavy use, while others
are disappointed, reporting only 4 hours of usage. Charging time
is a common complaint, though the actual battery performance
once charged is generally considered decent."

✓ Response is grounded in retrieved documents
✓ No hallucinated claims
```

Common Mistakes (and How to Avoid Them)

Mistake	Symptom	Fix
Chunks too large	Poor retrieval precision	Use 200-500 tokens per chunk
Chunks too small	Lost context	Add 50-100 token overlap
No overlap	Ideas split across chunks	Use overlapping windows
Too few results retrieved	Missing relevant info	Retrieve top 5-10 chunks
Too many results retrieved	Confuses the LLM	Limit to 3-5 most relevant
Not citing sources	Can't verify accuracy	Include chunk IDs in prompt
Ignoring retrieval failures	Wrong answers	Add "I don't know" fallback

If you see this error:

```
FAISS index error: dimension mismatch
```

Fix: Ensure query and document embeddings use the same model.

If RAG responses seem wrong: - Check retrieval first: Are the right chunks being retrieved? - Print the augmented prompt: Is context being passed correctly? - Test the LLM alone: Does it work without RAG?

Questions to Answer

- **Q1:** How did you decide on chunk size and overlap?
- **Q2:** Show an example where RAG improved the LLM's answer.
- **Q3:** Show an example where retrieval failed. Why?
- **Q4:** How would you deploy this for a real business application?

Going Deeper (Optional Challenges)

Challenge A: Hybrid Search

Combine semantic search (embeddings) with keyword search (BM25). Compare results: when does each approach win?

Challenge B: Re-ranking

After initial retrieval, use a cross-encoder to re-rank results. Does this improve answer quality?

Challenge C: Multi-hop RAG

Build a RAG system that can answer questions requiring information from multiple documents. Example: "Which product has the best battery life AND lowest price?"

Quick Reference

```python
# Install dependencies
!pip install sentence-transformers faiss-cpu langchain openai

# 1. CHUNK DOCUMENTS
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    separators=["\n\n", "\n", ". ", " ", ""]
)
chunks = splitter.split_documents(documents)

# 2. CREATE EMBEDDINGS
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
chunk_texts = [c.page_content for c in chunks]
embeddings = model.encode(chunk_texts)

# 3. BUILD VECTOR INDEX (FAISS)
import faiss
import numpy as np

dimension = embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(np.array(embeddings).astype('float32'))

# 4. RETRIEVE RELEVANT CHUNKS
def retrieve(query, k=5):
    query_emb = model.encode([query])
    distances, indices = index.search(
        np.array(query_emb).astype('float32'), k
```

```
    )
    return [chunks[i] for i in indices[0]]

# 5. BUILD RAG PROMPT
def build_prompt(query, retrieved_chunks):
    context = "\n\n".join([c.page_content for c in retrieved_chunks])
    return f"""Answer the question based on the context below.

Context:
{context}

Question: {query}

Answer:"""

# 6. GENERATE WITH LLM
from openai import OpenAI
client = OpenAI()

def rag_answer(query):
    chunks = retrieve(query)
    prompt = build_prompt(query, chunks)

    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content

# Alternative: Use HuggingFace model (free, no API key)
from transformers import pipeline
generator = pipeline("text-generation", model="google/flan-t5-base")
```

RAG Architecture Components: | Component | Purpose | Common Tools | |———–|———|————–| | Chunker | Split docs into pieces | LangChain, LlamaIndex | | Embedder | Convert text to vectors | Sentence Transformers | | Vector Store | Index and search | FAISS, Chroma, Pinecone | | Retriever | Find relevant chunks | Similarity search | | Generator | Create final answer | OpenAI, HuggingFace |

Submission

Upload to Canvas: - Your completed `.ipynb` notebook with all cells executed

Richard Young, Ph.D.