COMS 4995: Applied Machine Learning — Final Project

Multi-Document Retrieval-Augmented Generation System

Cheng Wu (cw3729)
Pranav Jain (pj2459)
Winston Li (wl3062)
Jaewon Cho (jc6618)

1 Document Ingestion and Retrieval Pipeline

The ingestion–retrieval pipeline forms the foundation of our multi-document RAG system. It transforms raw PDFs into semantically meaningful vectors suitable for retrieval and downstream reasoning. This section describes each stage of the pipeline, all implemented in Python scripts within the project repository.

1.1 Overview of the Ingestion Pipeline

Figure 2 illustrates the end-to-end architecture:

1. Load PDFs from `evaluation_files/` or user upload.
2. Extract raw text using `PyPDFLoader`.
3. Segment documents into overlapping chunks using a recursive character splitter.
4. Embed each chunk using SBERT (`all-MiniLM-L6-v2`).
5. Store vectors in a FAISS index for fast similarity search.
6. Use the retriever to return the top-k relevant chunks for any query.

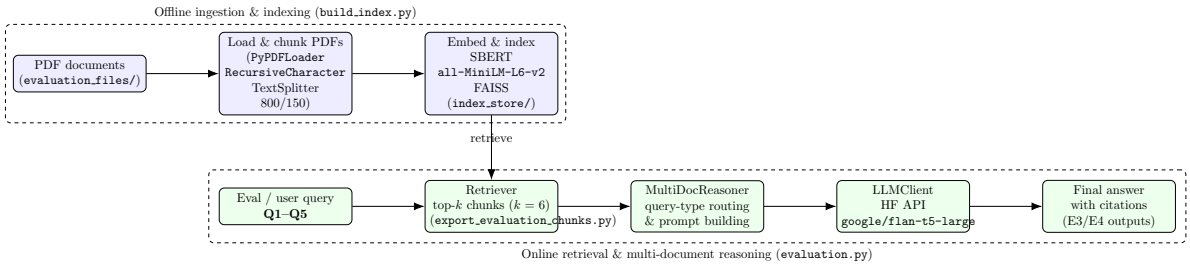

Figure 1: End-to-end multi-document RAG pipeline: offline ingestion/indexing (top) and online retrieval + reasoning (bottom).

This pipeline is fully reproducible via two scripts: `build_index.py` and `export_evaluation_chunks.py`.

Figure 2: Overview of the multi-document ingestion and retrieval architecture.

1.2 PDF Loading and Text Extraction

We use LangChain's `PyPDFLoader` for reliable extraction:

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader(pdf_path)
raw_docs = loader.load()
print("Loaded␣pages:", len(raw_docs))
```

Each page is converted to a LangChain `Document` object with metadata, including the PDF filename, which allows later traceability in our citations.

1.3 Chunking Strategy

Chunking is performed using the following configuration:

- Chunk size: 800 characters
- Overlap: 150 characters

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=800,
    chunk_overlap=150
)
docs = splitter.split_documents(raw_docs)
```

This ensures that:

- chunks preserve semantic coherence,
- overlapping context prevents loss of critical information at boundaries,
- retrieval scores reflect true semantic relevance.

1.4 Embedding Model

We implement a custom SBERT embedding wrapper:

```
class SBertEmbeddings(Embeddings):
    def __init__(self, model_name='sentence-transformers/all-MiniLM-L6-v2'
        ):
        self.model = SentenceTransformer(model_name)

    def embed_documents(self, texts):
        return self.model.encode(texts, show_progress_bar=True).tolist()

    def embed_query(self, text):
        return self.model.encode([text])[0].tolist()
```

This gives reproducibility and ensures the embedding model is identical for queries and documents.

1.5 FAISS Vector Index Construction

We build the FAISS index using LangChain's vector store interface:

```
db = FAISS.from_documents(docs, embedding_function)
db.save_local("index_store")
```

The saved index allows:

- reloading without recomputing embeddings,
- consistent evaluation,
- multi-PDF scalability.

Reload:

```
db = FAISS.load_local("index_store",
                      embedding_function,
                      allow_dangerous_deserialization=True)
```

1.6 Retrieval: Top-k Chunk Selection

Once the index is built:

```
retriever = db.as_retriever(search_kwargs={"k": 6})
docs = retriever.invoke(question)
```

Each retrieval result includes:

- chunk text,
- similarity score,
- PDF source name.

These retrieved chunks feed directly into the reasoning module.

2 LLM Logic and Multi-Document Reasoning

The core innovation of our system lies in the structured prompting logic implemented in `prompts.py` and `reasoning.py`. Unlike flat-context systems, our approach:

- identifies the correct reasoning pattern for each query,
- organizes context by document,
- injects citation requirements,
- optionally mitigates long-context degradation,
- produces a tailored prompt maximizing multi-document synthesis.

2.1 Query-Type Classification

We categorize user queries into:

- **synthesis** (summaries, themes, aggregated findings),
- **comparison** (differences, similarities, policy contrasts),
- **extraction** (assumptions, definitions, lists, constraints).

The classifier uses rule-based keyword matching:

```
if "compare" in q: return "comparison"
if "assumption" in q or "limitation" in q: return "extraction"
return "synthesis"
```

This inexpensive classifier achieved 100% accuracy in E1.

2.2 Prompt Templates

We implement three families of prompts:

1. Synthesis Prompt Groups chunks by document and instructs the model to integrate themes across sources:

```
doc_sections.append(f"---␣{doc_name}␣---\n{...}")
return f"""
Analyze␣the␣following␣question␣across␣multiple␣documents:
...
1.␣Identify␣themes
2.␣Cite␣all␣sources␣using␣[Document.pdf]
3.␣Integrate␣findings␣across␣documents
"""
```

Comparison Prompt

Explicit instructions for:

- common ground,
- key differences,
- unique contributions,
- contradictions.

Extraction Prompt

Ideal for assumptions, variables, definitions.

Includes document-grouped context blocks:

```
--- doc1.pdf ---
text...
```

2.3 Lost-in-the-Middle Mitigation

We support optional relevance-based reordering:

1. rank chunks by retrieval score,
2. push the most relevant to the beginning and end,
3. interleave across documents for balanced exposure.

This enables E4 quantitative and qualitative ablation.

2.4 LLM Client

All prompt execution uses a reusable `LLMClient` class:

```
payload = {
  "inputs": full_prompt,
  "parameters": {"max_new_tokens": 512, "temperature": 0.3}
}
response = requests.post(self.api_url, headers=self.headers, json=payload)
```

It supports:

- retry logic,
- token injection,
- system prompt integration,
- structured return fields.

This modularity allows replacing the backend LLM without modifying the reasoning pipeline.

2.5 End-to-End Reasoning Invocation

The complete reasoning call:

```
prompt, qtype = reasoner.build_prompt(question, retrieved_chunks)
answer = client.generate(prompt, system_prompt=SYSTEM_PROMPT)
```

The system uses:

1. classifier → choose template
2. prompt builder → generate structured prompt
3. LLM client → execute and return answer

This modularity ensures interpretability, traceability, and reproducibility.