RAG & Vector Databases - Lab

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Lab Overview

Goal: Build a production-ready "Chat with PDF" system using RAG.

What You'll Build:

1. PDF ingestion pipeline with intelligent chunking
2. Vector database with ChromaDB
3. Semantic search with embeddings
4. LLM-powered Q&A system
5. Interactive Streamlit interface

Learning Objectives:

- Understand embedding mathematics
- Implement chunking strategies
- Build retrieval systems
- Evaluate RAG performance

Prerequisites

Knowledge:

- Python programming
- Basic understanding of vectors
- Familiarity with LLMs

API Keys:

- Gemini API key (free from https://makersuite.google.com/app/apikey)
- OR OpenAI API key

Sample PDFs:

- Research paper (arXiv)
- Course syllabus
- Technical documentation

Setup: Environment

Step 1: Create project directory:

```
mkdir rag-lab
cd rag-lab
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
```

Step 2: Install dependencies:

```
pip install \
  chromadb \
  sentence-transformers \
  pypdf \
  langchain \
  google-generativeai \
  streamlit \
  numpy \
  pandas
```

Part 1: Understanding Embeddings

Create `embeddings_demo.py`:

```
from sentence_transformers import SentenceTransformer
import numpy as np
from numpy.linalg import norm

# Load model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Create embeddings
texts = [
    "The cat sits on the mat",
    "A feline rests on the rug",
    "The dog plays in the yard",
    "Machine learning is a subset of AI"
]

embeddings = model.encode(texts)

print(f"Embedding shape: {embeddings[0].shape}")  # (384,)
print(f"First 5 dimensions: {embeddings[0][:5]}")
```

Part 1: Cosine Similarity Implementation

Add to `embeddings_demo.py`:

```
def cosine_similarity(a, b):
    """
    Compute cosine similarity between two vectors.
    Formula: cos(θ) = (a·b) / (||a|| ||b||)
    """
    return np.dot(a, b) / (norm(a) * norm(b))

# Compute similarities
print("\nSimilarity Matrix:")
for i, text1 in enumerate(texts):
    for j, text2 in enumerate(texts):
        if i < j:  # Only upper triangle
            sim = cosine_similarity(embeddings[i], embeddings[j])
            print(f"{text1[:30]:30} ⟷ {text2[:30]:30} = {sim:.3f}")
```

Expected output:

```
cat/mat ⟷ feline/rug        = 0.721  # High (synonyms)
```

Part 1: Vector Space Visualization

Add visualization (optional but recommended):

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Reduce to 2D for visualization
pca = PCA(n_components=2)
embeddings_2d = pca.fit_transform(embeddings)

plt.figure(figsize=(10, 6))
plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1], s=100)

for i, txt in enumerate(texts):
    plt.annotate(txt[:20], (embeddings_2d[i, 0], embeddings_2d[i, 1]))

plt.title("Text Embeddings in 2D Space")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.savefig("embeddings_viz.png")
plt.show()
```

Part 2: PDF Ingestion

Create `pdf_processor.py`:

```
from pypdf import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from typing import List, Dict

def extract_text_from_pdf(pdf_path: str) -> str:
    """Extract all text from PDF."""
    reader = PdfReader(pdf_path)
    text = ""

    for page_num, page in enumerate(reader.pages):
        page_text = page.extract_text()
        text += f"\n--- Page {page_num + 1} ---\n{page_text}"

    return text

def chunk_text(text: str, chunk_size: int = 500, overlap: int = 50) -> List[Dict]:
    """Split text into overlapping chunks."""
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=overlap,
        separators=["\n\n", "\n", ". ", " ", ""]
    )

    chunks = splitter.split_text(text)

    # Add metadata
    chunk_dicts = []
    for i, chunk in enumerate(chunks):
        chunk_dicts.append({
            "text": chunk,
            "chunk_id": i,
            "char_count": len(chunk),
            "word_count": len(chunk.split())
        })

    return chunk_dicts
```

Part 2: Testing PDF Processing

Create `test_pdf_processing.py`:

```python
from pdf_processor import extract_text_from_pdf, chunk_text

# Test with your PDF
pdf_path = "sample_paper.pdf"  # Replace with your PDF

print("Extracting text from PDF...")
text = extract_text_from_pdf(pdf_path)
print(f"Total characters: {len(text):,}")
print(f"Total words: {len(text.split()):,}")

print("\nChunking text...")
chunks = chunk_text(text, chunk_size=500, overlap=50)
print(f"Number of chunks: {len(chunks)}")

# Show first chunk
print("\n--- First Chunk ---")
print(f"ID: {chunks[0]['chunk_id']}")
print(f"Words: {chunks[0]['word_count']}")
print(f"Text preview:\n{chunks[0]['text'][:200]}...")

# Chunk statistics
import pandas as pd
df = pd.DataFrame(chunks)
print("\nChunk Statistics:")
print(df[['chunk_id', 'char_count', 'word_count']].describe())
```

Part 3: ChromaDB Setup

Create `vector_store.py`:

```python
import chromadb
from chromadb.config import Settings
from sentence_transformers import SentenceTransformer
from typing import List, Dict
import uuid

class VectorStore:
    def __init__(self, collection_name: str = "pdf_docs", persist_dir: str = "./chroma_db"):
        """Initialize ChromaDB with persistent storage."""
        self.client = chromadb.PersistentClient(path=persist_dir)

        # Create or get collection
        self.collection = self.client.get_or_create_collection(
            name=collection_name,
            metadata={"description": "PDF document chunks"}
        )

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

    def add_documents(self, chunks: List[Dict]):
        """Add document chunks to vector store."""
        texts = [chunk['text'] for chunk in chunks]
        ids = [str(uuid.uuid4()) for _ in chunks]

        # Generate embeddings
        embeddings = self.embedding_model.encode(texts).tolist()

        # Prepare metadata
        metadatas = [
            {
                "chunk_id": str(chunk['chunk_id']),
                "char_count": chunk['char_count'],
                "word_count": chunk['word_count']
            }
            for chunk in chunks
        ]

        # Add to collection
        self.collection.add(
            documents=texts,
            embeddings=embeddings,
            metadatas=metadatas,
            ids=ids
        )

        print(f"✅ Added {len(chunks)} chunks to vector store")
```

Part 3: ChromaDB Operations (Continued)

```python
    def search(self, query: str, n_results: int = 3) -> Dict:
        """Search for relevant chunks."""
        # Embed query
        query_embedding = self.embedding_model.encode([query])[0].tolist()

        # Search
        results = self.collection.query(
            query_embeddings=[query_embedding],
            n_results=n_results
        )

        return results

    def get_stats(self) -> Dict:
        """Get collection statistics."""
        count = self.collection.count()
        return {
            "total_chunks": count,
            "collection_name": self.collection.name
        }
```

Part 3: Testing Vector Store

Create `test_vector_store.py` :

```python
from pdf_processor import extract_text_from_pdf, chunk_text
from vector_store import VectorStore

# 1. Process PDF
pdf_path = "sample_paper.pdf"
text = extract_text_from_pdf(pdf_path)
chunks = chunk_text(text)

# 2. Initialize vector store
vs = VectorStore(collection_name="my_paper")

# 3. Add documents
vs.add_documents(chunks)

# 4. Test search
query = "What is the main contribution of this paper?"
results = vs.search(query, n_results=3)

print(f"\nQuery: {query}")
print(f"\nTop {len(results['documents'][0])} results:")

for i, doc in enumerate(results['documents'][0]):
    print(f"\n—— Result {i+1} ——")
    print(f"Text: {doc[:200]}...")
    print(f"Metadata: {results['metadatas'][0][i]}")
```

Part 4: RAG Pipeline

Create `rag_pipeline.py`:

```python
import google.generativeai as genai
from vector_store import VectorStore
from typing import Dict, List

class RAGPipeline:
    def __init__(self, vector_store: VectorStore, api_key: str):
        """Initialize RAG pipeline."""
        self.vs = vector_store

        # Configure Gemini
        genai.configure(api_key=api_key)
        self.llm = genai.GenerativeModel('gemini-pro')

    def retrieve(self, query: str, top_k: int = 3) -> Dict:
        """Retrieve relevant contexts."""
        results = self.vs.search(query, n_results=top_k)

        return {
            "documents": results['documents'][0],
            "metadatas": results['metadatas'][0],
            "distances": results.get('distances', [[]])[0]
        }

    def augment_prompt(self, query: str, contexts: List[str]) -> str:
        """Create augmented prompt with retrieved context."""
        context_text = "\n\n".join([
            f"[Context {i+1}]\n{ctx}"
            for i, ctx in enumerate(contexts)
        ])

        prompt = f"""You are a helpful AI assistant. Answer the question based ONLY on the provided context. If the answer is not in the context, say "I don't have enough information to answer this question."

Context:
{context_text}

Question: {query}

Answer:"""

        return prompt
```

Part 4: RAG Pipeline (Continued)

```python
    def generate(self, query: str, top_k: int = 3, show_sources: bool = True) -> Dict:
        """Full RAG pipeline: Retrieve + Augment + Generate."""
        # 1. Retrieve
        retrieval = self.retrieve(query, top_k)
        contexts = retrieval['documents']

        # 2. Augment prompt
        prompt = self.augment_prompt(query, contexts)

        # 3. Generate answer
        response = self.llm.generate_content(prompt)
        answer = response.text

        result = {
            "query": query,
            "answer": answer,
            "num_contexts": len(contexts)
        }

        if show_sources:
            result["sources"] = [
                {
                    "text": ctx[:200] + "...",
                    "metadata": meta
                }
                for ctx, meta in zip(contexts, retrieval['metadatas'])
            ]

        return result
```

Part 4: Testing RAG Pipeline

Create `test_rag.py` :

```python
from pdf_processor import extract_text_from_pdf, chunk_text
from vector_store import VectorStore
from rag_pipeline import RAGPipeline
import os

# 1. Setup (only once)
pdf_path = "sample_paper.pdf"
text = extract_text_from_pdf(pdf_path)
chunks = chunk_text(text)

vs = VectorStore(collection_name="rag_demo")
vs.add_documents(chunks)

# 2. Initialize RAG
api_key = os.getenv("GEMINI_API_KEY")  # Set this in environment
rag = RAGPipeline(vs, api_key)

# 3. Ask questions
questions = [
    "What is the main contribution of this paper?",
    "What datasets were used in the experiments?",
    "What are the limitations mentioned?"
]

for q in questions:
    print(f"\n{'='*60}")
    print(f"Q: {q}")
    print(f"{'='*60}")

    result = rag.generate(q, top_k=3)

    print(f"\nA: {result['answer']}")
    print(f"\n[Used {result['num_contexts']} context chunks]")
```

Part 5: Evaluation Metrics

Create `rag_evaluator.py`:

```
import numpy as np
from typing import List, Set

class RAGEvaluator:
    @staticmethod
    def recall_at_k(relevant_ids: Set[int], retrieved_ids: List[int], k: int) -> float:
        """
        Compute Recall@K.
        Formula: |relevant ∩ top-K| / |relevant|
        """
        top_k = set(retrieved_ids[:k])
        intersection = relevant_ids.intersection(top_k)

        if len(relevant_ids) == 0:
            return 0.0

        return len(intersection) / len(relevant_ids)

    @staticmethod
    def mean_reciprocal_rank(relevant_ids: Set[int], retrieved_ids: List[int]) -> float:
        """
        Compute MRR.
        Formula: 1 / rank of first relevant doc
        """
        for rank, doc_id in enumerate(retrieved_ids, start=1):
            if doc_id in relevant_ids:
                return 1.0 / rank
        return 0.0

    @staticmethod
    def precision_at_k(relevant_ids: Set[int], retrieved_ids: List[int], k: int) -> float:
        """
        Compute Precision@K.
        Formula: |relevant ∩ top-K| / K
        """
        top_k = set(retrieved_ids[:k])
        intersection = relevant_ids.intersection(top_k)

        return len(intersection) / k
```

Part 5: Testing Evaluation

Create `test_evaluation.py`:

```python
from rag_evaluator import RAGEvaluator

# Simulated scenario
relevant_docs = {2, 5, 8}  # Ground truth: doc IDs 2, 5, 8 are relevant
retrieved_docs = [2, 10, 5, 15, 8, 20]  # Retrieval system output

evaluator = RAGEvaluator()

# Metrics
recall_3 = evaluator.recall_at_k(relevant_docs, retrieved_docs, k=3)
recall_5 = evaluator.recall_at_k(relevant_docs, retrieved_docs, k=5)
mrr = evaluator.mean_reciprocal_rank(relevant_docs, retrieved_docs)
precision_3 = evaluator.precision_at_k(relevant_docs, retrieved_docs, k=3)

print("Evaluation Results:")
print(f"Recall@3:     {recall_3:.3f}")  # 2/3 = 0.667 (found 2 out of 3 in top-3)
print(f"Recall@5:     {recall_5:.3f}")  # 3/3 = 1.0   (found all 3 in top-5)
print(f"MRR:          {mrr:.3f}")       # 1/1 = 1.0   (first result is relevant)
print(f"Precision@3:  {precision_3:.3f}")  # 2/3 = 0.667 (2 relevant in top-3)
```

Part 6: Streamlit UI

Create `app.py`:

```python
import streamlit as st
from pdf_processor import extract_text_from_pdf, chunk_text
from vector_store import VectorStore
from rag_pipeline import RAGPipeline
import os
import tempfile

st.set_page_config(page_title="Chat with PDF", page_icon="📄", layout="wide")

st.title("📄 Chat with Your PDF")
st.markdown("Upload a PDF and ask questions about its content")

# Sidebar for API key and settings
with st.sidebar:
    st.header("⚙️ Configuration")

    api_key = st.text_input(
        "Gemini API Key",
        type="password",
        help="Get from https://makersuite.google.com/app/apikey"
    )

    st.markdown("---")

    chunk_size = st.slider("Chunk Size", 100, 1000, 500)
    chunk_overlap = st.slider("Chunk Overlap", 0, 200, 50)
    top_k = st.slider("Number of contexts", 1, 10, 3)

    show_sources = st.checkbox("Show source chunks", value=True)
```

Part 6: Streamlit UI (Continued)

```python
# File upload
uploaded_file = st.file_uploader("Upload PDF", type=['pdf'])

if uploaded_file and api_key:
    # Save uploaded file temporarily
    with tempfile.NamedTemporaryFile(delete=False, suffix='.pdf') as tmp_file:
        tmp_file.write(uploaded_file.getvalue())
        tmp_path = tmp_file.name

    # Initialize session state
    if 'processed' not in st.session_state or st.session_state.get('last_file') != uploaded_file.name:
        with st.spinner("Processing PDF..."):
            # Extract and chunk
            text = extract_text_from_pdf(tmp_path)
            chunks = chunk_text(text, chunk_size, chunk_overlap)

            # Create vector store
            vs = VectorStore(collection_name="temp_pdf")
            vs.add_documents(chunks)

            # Initialize RAG
            st.session_state.rag = RAGPipeline(vs, api_key)
            st.session_state.processed = True
            st.session_state.last_file = uploaded_file.name
            st.session_state.chunk_count = len(chunks)

        st.success(f"✅ Processed {st.session_state.chunk_count} chunks")
```

Part 6: Streamlit UI (Chat Interface)

```python
    # Chat interface
    st.header("💬 Ask Questions")

    # Initialize chat history
    if 'messages' not in st.session_state:
        st.session_state.messages = []

    # Display chat history
    for message in st.session_state.messages:
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

            if message["role"] == "assistant" and "sources" in message:
                with st.expander("View Sources"):
                    for i, src in enumerate(message["sources"]):
                        st.markdown(f"**Source {i+1}:**")
                        st.text(src["text"])
                        st.caption(f"Chunk ID: {src['metadata']['chunk_id']}")

    # User input
    if prompt := st.chat_input("Ask a question about the PDF..."):
        # Add user message
        st.session_state.messages.append({"role": "user", "content": prompt})

        with st.chat_message("user"):
            st.markdown(prompt)

        # Generate response
        with st.chat_message("assistant"):
            with st.spinner("Thinking..."):
                result = st.session_state.rag.generate(
                    prompt,
                    top_k=top_k,
                    show_sources=show_sources
                )

                st.markdown(result["answer"])

                message_data = {
                    "role": "assistant",
                    "content": result["answer"]
                }

                if show_sources and "sources" in result:
                    message_data["sources"] = result["sources"]

                    with st.expander("View Sources"):
                        for i, src in enumerate(result["sources"]):
                            st.markdown(f"**Source {i+1}:**")
                            st.text(src["text"])
                            st.caption(f"Chunk ID: {src['metadata']['chunk_id']}")

                st.session_state.messages.append(message_data)
elif not api_key:
    st.warning("👈 Please enter your Gemini API key in the sidebar")
elif not uploaded_file:
    st.info("👆 Please upload a PDF to get started")
```

Part 6: Running the App

Run Streamlit:

```
streamlit run app.py
```

Test the app:

1. Enter your Gemini API key in sidebar
2. Upload a PDF (e.g., research paper)
3. Wait for processing
4. Ask questions:
   - "What is this document about?"
   - "What are the main findings?"
   - "What methodology was used?"
5. Check "Show source chunks" to see retrieved context

Part 7: Advanced Features

Create `hybrid_search.py` for keyword + semantic search:

```python
from rank_bm25 import BM25Okapi
import numpy as np

class HybridSearch:
    def __init__(self, vector_store, documents):
        self.vs = vector_store
        self.documents = documents

        # Build BM25 index
        tokenized_docs = [doc.lower().split() for doc in documents]
        self.bm25 = BM25Okapi(tokenized_docs)

    def search(self, query: str, top_k: int = 5, alpha: float = 0.5):
        """
        Hybrid search: combine vector (semantic) + BM25 (keyword).

        alpha: weight for vector search (1-alpha for BM25)
        """
        # 1. Vector search
        vector_results = self.vs.search(query, n_results=len(self.documents))
        vector_scores = 1.0 - np.array(vector_results.get('distances', [[]])[0])

        # 2. BM25 search
        tokenized_query = query.lower().split()
        bm25_scores = self.bm25.get_scores(tokenized_query)

        # Normalize scores
        vector_scores = vector_scores / (vector_scores.max() + 1e-8)
        bm25_scores = bm25_scores / (bm25_scores.max() + 1e-8)

        # Combine
        final_scores = alpha * vector_scores + (1 - alpha) * bm25_scores

        # Get top-K
        top_indices = np.argsort(final_scores)[::-1][:top_k]

        return [self.documents[i] for i in top_indices]
```

Part 7: Re-ranking

Add re-ranker for better precision:

```python
from sentence_transformers import CrossEncoder

class Reranker:
    def __init__(self):
        # Load cross-encoder model
        self.model = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

    def rerank(self, query: str, documents: List[str], top_k: int = 3) -> List[str]:
        """
        Re-rank documents using cross-encoder.
        More accurate but slower than bi-encoder.
        """
        # Score all query-document pairs
        pairs = [[query, doc] for doc in documents]
        scores = self.model.predict(pairs)

        # Sort by score
        sorted_indices = np.argsort(scores)[::-1]

        return [documents[i] for i in sorted_indices[:top_k]]

# Usage in RAG pipeline:
# 1. Retrieve 50 candidates (fast bi-encoder)
# 2. Re-rank top 50 (slower cross-encoder)
# 3. Use top 3 for LLM context
```

Part 8: Complete Project Structure

```
rag-lab/
├── app.py                  # Streamlit UI
├── embeddings_demo.py      # Part 1: Embeddings
├── pdf_processor.py        # Part 2: PDF processing
├── vector_store.py         # Part 3: ChromaDB
├── rag_pipeline.py         # Part 4: RAG logic
├── rag_evaluator.py        # Part 5: Metrics
├── hybrid_search.py        # Part 7: Advanced search
├── test_*.py               # Test scripts
├── chroma_db/              # Persistent vector DB
├── requirements.txt
└── README.md
```

requirements.txt:

```
chromadb==0.4.18
sentence-transformers==2.2.2
pypdf==3.17.0
langchain==0.1.0
google-generativeai==0.3.1
```

Exercises

Exercise 1: Experiment with Chunk Size

- Try chunk sizes: 200, 500, 1000
- Measure retrieval quality for each
- Plot chunk size vs Recall@3

Exercise 2: Compare Embedding Models

- Try `all-MiniLM-L6-v2` (fast, 384 dim)
- Try `all-mpnet-base-v2` (better, 768 dim)
- Compare cosine similarities

Exercise 3: Build a Multi-Document System

- Ingest 3 different PDFs
- Add document metadata (title, author)
- Filter search by document

Debugging Common Issues

Issue 1: "Model too large for memory"

- **Solution:** Use smaller model (`all-MiniLM-L6-v2`)
- Or: Use API-based embeddings (OpenAI, Gemini)

Issue 2: "PDF text extraction garbled"

- **Solution:** Try different PDF libraries (pdfplumber, PyMuPDF)
- Or: Pre-process with OCR (tesseract)

Issue 3: "Retrieval returns irrelevant chunks"

- **Solution:**
  - Adjust chunk size/overlap
  - Try hybrid search (keyword + semantic)
  - Use re-ranking

Issue 4: "LLM hallucinates despite context"

Testing Checklist

Functionality:

- [ ] PDF upload and text extraction works
- [ ] Chunking produces reasonable segments
- [ ] Embeddings are generated correctly
- [ ] Vector search returns relevant results
- [ ] RAG pipeline generates accurate answers
- [ ] Streamlit UI is responsive

Quality:

- [ ] Recall@3 > 0.6 for known questions
- [ ] MRR > 0.7 for first relevant result
- [ ] LLM answers are factually correct
- [ ] No hallucinations detected

Submission Requirements

Deliverables:

1. Complete code (all Python files)
2. Sample PDF used for testing
3. README with:
    - Installation instructions
    - Usage examples
    - Screenshots of Streamlit app
4. Evaluation results:
    - Recall@K for 5 test questions
    - Comparison of 2+ chunk sizes

Bonus Points (+10 each):

- Implement hybrid search
- Add re-ranking

Grading Rubric

Component	Points	Criteria
PDF Processing	15	Correct text extraction and chunking
Vector Store	20	ChromaDB integration, embeddings
RAG Pipeline	25	Retrieval + generation working
Streamlit UI	20	Functional, user-friendly interface
Evaluation	10	Metrics implementation
Documentation	10	Clear README, comments
Bonus	+30	Advanced features

Total: 100 points (+30 bonus)

Resources

Documentation:

- ChromaDB: https://docs.trychroma.com/
- Sentence Transformers: https://www.sbert.net/
- LangChain: https://python.langchain.com/

Tutorials:

- Pinecone Learning: https://www.pinecone.io/learn/
- DeepLearning.AI: "Building Systems with LLM API"

Research Papers:

- "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (Lewis et al., 2020)
- "Dense Passage Retrieval for Open-Domain QA" (Karpukhin et al., 2020)

Questions?

Office Hours: [Time/Location]

Email: [instructor email]

Discussion Forum: [link]

Happy Building! 🚀