partnr

Build a Multimodal RAG System for Document and Image Analysis

Back

Mandatory Task

Domain

Data Engineering

Machine Learning

Skills

API Development | Artificial Intelligence | Computer Vision | Database Management | Machine Learning | System Design | Optical Character Recognition | Retrieval-Augmented Generation

Difficulty

Hard

Tools

Android Debug Bridge | FAISS | FastAPI | Flask | LLaVA

Report Issue

textual data simultaneously, a critical capability for industries like research, finance, and

partnr

sophisticated data ingestion pipelines involving Optical Character Recognition (OCR) and table extraction to implementing cutting-edge multimodal embeddings. You will design and build a cross-modal retrieval system that can find relevant images from text queries and vice-versa. The final system's quality will be judged on its ability to generate accurate, visually-grounded answers by integrating with a powerful Vision-Language Model (VLM), demonstrating a deep understanding of production-level AI engineering.

Core Requirements

Data Ingestion and Processing

- The system must support ingestion of PDF documents, PNG/JPEG images, and plain text files.
- Implement an Optical Character Recognition (OCR) pipeline to extract text from images and embedded images within PDFs.
- Include functionality to detect and extract tabular data from documents, preserving their structure (rows and columns).
- The processing pipeline should handle complex layouts, such as multi-column scientific papers or presentation slides.

Multimodal Embeddings and Indexing

- Generate meaningful vector embeddings for different content types: text chunks, images, and potentially summaries of tables.
- Utilize a multimodal embedding model (e.g., CLIP-based) capable of placing text and images in a shared semantic space.
- Create a unified indexing strategy in a vector database (e.g., ChromaDB, FAISS) that stores embeddings and associated metadata for all content types.
- The metadata for each vector must include the source document, page number, and content type (text, image, table).

Cross-Modal Retrieval

- Design and implement a retrieval system that can accept a text query and return a ranked list of relevant text chunks, images, and tables.
- The retriever must support cross-modal search, meaning a text query can retrieve relevant images, and an image query (if implemented) can retrieve relevant text.
- Implement a fusion or re-ranking strategy to combine results from different modalities into a single, coherent context for the generator.

Generation with Visual Grounding

- Integrate a Vision-Language Model (VLM) such as GPT-4V (via API) or an open-source alternative like LLaVA for the generation step.

- The system must be able to pass both text and image data (e.g., image paths or

partnr

image, the text should reference the visual element (e.g., "As shown in the bar chart on page 5...").

API and Functionality

- Expose the RAG system's functionality through a REST API built with a web framework like FastAPI or Flask.
- The API must have at least one endpoint that accepts a user query (text) and returns a comprehensive answer in JSON format, including the generated text and source references.

Implementation Guidelines

Recommended Technology Stack

- **Language:** Python is required.
- **Document Processing:** Libraries like `unstructured`, `PyMuPDF` for parsing, and `pytesseract` for OCR are highly recommended.
- **Image Handling:** Use `Pillow` or `OpenCV` for image manipulation and preprocessing.
- **Embeddings:** A CLIP-based model from a library like `sentence-transformers` is a strong starting point.
- **Vector Database:** An in-memory or local file-based vector DB like `ChromaDB` or `FAISS` is sufficient for this project.
- **VLM:** Utilize a powerful VLM. The OpenAI API for GPT-4V is a straightforward choice, but open-source models like LLaVA are also excellent alternatives.
- **API:** `FastAPI` is recommended for its performance and ease of use.

Architectural Design

- **Decoupled Pipeline:** Design your system as a series of decoupled stages: Ingestion, Chunking, Embedding, Indexing, Retrieval, and Generation. This modularity simplifies development and testing.
- **Hybrid Indexing:** Consider creating separate indexes or namespaces within your vector DB for text and images. Your retriever can query both and then fuse the results.
- **Metadata is Key:** Store rich metadata alongside your vectors. This should include document ID, page number, content type, and for images, a file path or identifier to retrieve the raw image data for the VLM.

Key Challenges to Consider

- **Chunking Strategy:** How do you create meaningful chunks from documents that mix text and images? Should an image be its own chunk, or should it be associated with surrounding text?
- **Context Formulation:** Constructing the prompt for the VLM is critical. You'll need a robust method to serialize both text snippets and images into a format the model can

understand.

partnr

- A fully functional multimodal RAG system accessible via a REST API.
- Demonstrated ability to ingest and process a collection of at least 10 diverse documents (PDFs with images, standalone images, etc.).
- The system successfully answers questions that require synthesizing information from both text and images within the documents.
- An evaluation notebook or script that measures retrieval performance (e.g., hit rate, MRR) on a small, curated set of multimodal questions.
- Source references returned with each answer are accurate, pointing to the correct document, page, and content type.
- The API endpoint for querying the system has a response latency of under 15 seconds for a typical query on the test document set.
- Comprehensive documentation covering the system's architecture, setup, and API usage.

Implementation Details

Step 1: Project Setup and Environment

Set up your project directory. Create a `requirements.txt` for Python dependencies. Ensure your environment can handle large files and potentially external API calls.

```
# Example project structure
.
├── src/
│   ├── api/
│   │   └── main.py
│   ├── ingestion/
│   │   └── document_parser.py
│   │   └── image_processor.py
│   ├── embeddings/
│   │   └── model_loader.py
│   ├── retrieval/
│   │   └── retriever.py
│   ├── generation/
│   │   └── generator.py
│   └── vector_store/
│       └── chroma_manager.py
├── tests/
│   ├── test_ingestion.py
│   └── test_api.py
├── sample_documents/ # Mandatory: Your test documents
├── ARCHITECTURE.md   # Mandatory: System architecture documentation
├── README.md         # Mandatory: Project setup and usage
├── requirements.txt
├── .env.example      # Mandatory: Example environment variables (e.g., VLM_API_KEY)
└── submission.yml    # Mandatory: Automated evaluation commands
```

Step 2: Document Ingestion Pipeline

Develop modules to handle different document types.

partnr

- Implement `image_processor.py` for standalone images (PNG/JPEG), performing OCR and potentially image feature extraction.
- Ensure extracted data includes metadata: original file, page number, bounding boxes (if applicable), and content type (text, table, image).

Step 3: Multimodal Embedding and Indexing

- Choose and integrate a multimodal embedding model (e.g., a CLIP-based model from `sentence-transformers`).
- Create a `vector_store` module (e.g., `chroma_manager.py`) to manage your vector database (ChromaDB or FAISS).
- For each extracted chunk (text, image feature, table summary), generate its vector embedding.
- Store the embedding along with rich metadata (source, page, content type, raw content reference) in your vector database.

Step 4: Cross-Modal Retrieval API

- Develop a `retriever.py` module that takes a text query.
- Query your vector database to retrieve relevant text chunks, image embeddings, and table embeddings.
- Implement a fusion or re-ranking logic to combine results from different modalities into a single, ranked list of contextual items. This could involve combining similarity scores or using a re-ranking model.

Step 5: VLM Integration and Response Generation

- In the `generation` module, integrate with your chosen Vision-Language Model (VLM). This will likely involve making API calls (e.g., to OpenAI GPT-4V) or loading an open-source model (e.g., LLaVA).
- Design a prompt engineering strategy to effectively pass both textual context (retrieved text, table summaries) and visual context (raw images retrieved) to the VLM.
- Ensure the VLM can generate responses that are "visually grounded" by referencing elements from the images when appropriate.

Step 6: API Endpoint Development

- Create a REST API using a framework like FastAPI or Flask (`src/api/main.py`).
- Implement an endpoint, e.g., `/query`, that accepts a POST request with a user's text query.
- This endpoint should orchestrate the retrieval and generation steps:
  1. Receive query.
  2. Call the retrieval system to get multimodal context.
  3. Format context (text + images) for the VLM.

4. Call the VLM for generation.

partnr

```
// Example API endpoint: POST /query
// Request Body:
{
  "query": "What is the key takeaway from the chart on page 3?"
}

// Response Body:
{
  "answer": "The bar chart on page 3 illustrates a significant increase in Q3 sales, reaching 1
  "sources": [
    {
      "document_id": "report_2023.pdf",
      "page_number": 3,
      "content_type": "image",
      "snippet": "path/to/image_page3_chart.png"
    },
    {
      "document_id": "report_2023.pdf",
      "page_number": 3,
      "content_type": "text",
      "snippet": "Q3 sales surged to 1.2 million units, exceeding projections due to strong dem
    }
  ]
}
```

Step 7: Automated Tests

- Develop a comprehensive test suite in the `tests/` directory.
- Include unit tests for individual components (parsers, embedders, retrievers).
- Crucially, create an end-to-end integration test that simulates document ingestion and querying the API for a multimodal answer. This test should be runnable via your `submission.yml`.

Step 8: Documentation

- Create a detailed `README.md` with setup instructions, how to run the application, and API usage examples.
- Develop an `ARCHITECTURE.md` that clearly explains your system's design, including diagrams for data flow and component interactions.
- Provide a `sample_documents/` directory with at least 10 diverse test documents (PDFs with images, standalone images, plain text).
- Include an `evaluation.ipynb` or script demonstrating your evaluation methodology and results.

Common Mistakes To Avoid

- **Ignoring Image Context:** Simply extracting text from images via OCR and discarding

partnr

About Us

Contact Us

partnr