AI VIET NAM
@aivietnam.edu.vn

AI VIET NAM – AI COURSE 2025

Tutorial: RAG

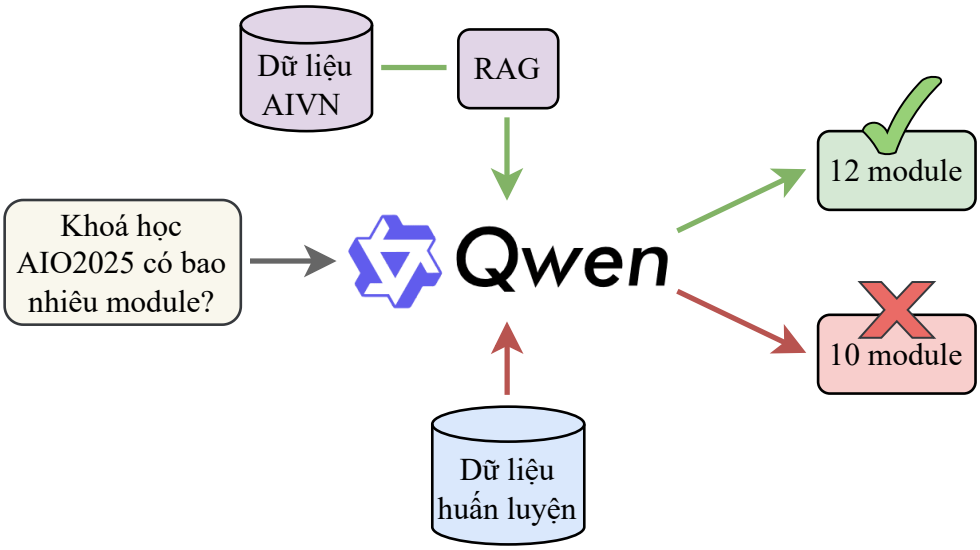
Dương Trường Bình Nguyễn Anh Khôi Trần Đại Nhân
Dương Đình Thắng Đinh Quang Vinh

I. Dẫn nhập

Sự bùng nổ của các Large Language Models (LLMs) như ChatGPT, Gemini, Claude hay Llama đã định hình lại lĩnh vực NLP. Tuy nhiên, dù sở hữu khả năng tổng quát và suy luận ấn tượng, các mô hình này vẫn đối mặt với những hạn chế cố hữu: tri thức bị giới hạn tại thời điểm huấn luyện, hiện tượng ảo giác khi gặp các câu hỏi nằm ngoài vùng tri thức, và đặc biệt là sự thiếu hụt kiến thức về dữ liệu riêng tư của doanh nghiệp.

Để giải quyết vấn đề này, kỹ thuật Retrieval Augmented Generation (RAG) đã ra đời. RAG cho phép LLMs tiếp cận nguồn dữ liệu bên ngoài mà không cần trải qua quá trình fine-tuning tốn kém hay phải huấn luyện lại. Để giới thiệu về RAG, bài viết này đi sâu vào kiến trúc và cách triển khai, bao gồm:

1. Phân tích sâu khái niệm, kiến trúc, pipeline cơ bản của RAG.
2. Giới thiệu framework LangChain, công cụ mạnh mẽ cho các ứng dụng sử dụng LLM.
3. Xây dựng hệ thống QA System trên tài liệu PDF học thuật.



Hình 1: Minh hoạ LLM khi có (đường xanh lá) và không sử dụng RAG (đường màu đỏ).

Mục lục

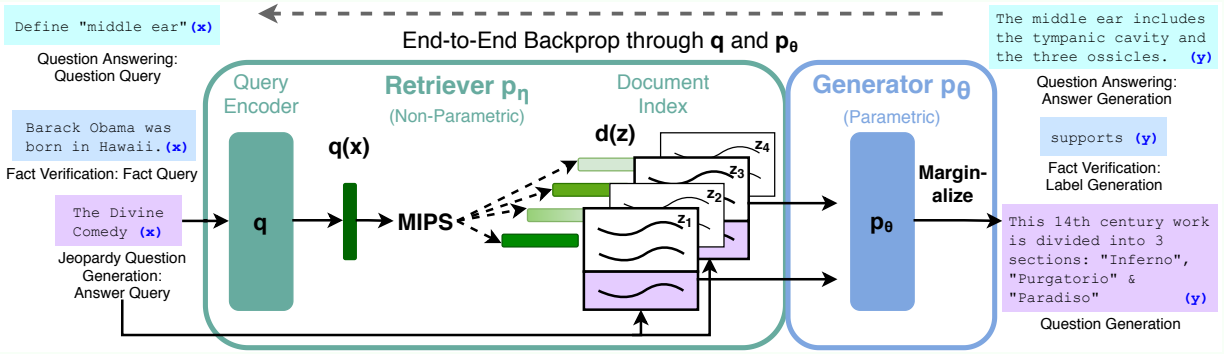
Bảng thuật ngữ

Thuật ngữ	Mô tả
Hallucination	Hiện tượng mô hình sinh ra thông tin sai lệch, bịa đặt hoặc không có thực nhưng với văn phong tự tin.
Knowledge Cutoff	Mốc thời gian giới hạn của dữ liệu huấn luyện, khiến mô hình không biết các sự kiện xảy ra sau đó.
Fine-tuning	Quá trình huấn luyện tiếp mô hình đã pre-trained trên tập dữ liệu chuyên biệt để cập nhật trọng số.
In-Context Learning	Khả năng LLM học và thực hiện tác vụ dựa trên ngữ cảnh hoặc ví dụ được cung cấp trong prompt mà không cần cập nhật tham số.
Vector Embeddings	Biểu diễn dữ liệu (text, image) dưới dạng vector số thực trong không gian n-chiều.
Semantic Search	Tìm kiếm dựa trên sự tương đồng về ý nghĩa thay vì chỉ khớp từ khóa.
Chunking	Kỹ thuật chia nhỏ văn bản dài thành các đoạn ngắn để tối ưu hóa việc mã hóa và phù hợp với giới hạn Context Window.
Context Window	Giới hạn số lượng tokens (đơn vị văn bản) tối đa mà LLM có thể tiếp nhận và xử lý trong một lần prompt.
Grounding	Kỹ thuật "neo" câu trả lời của mô hình vào dữ liệu thực tế được cung cấp để đảm bảo tính xác thực.

II. Cơ sở lý thuyết về RAG

Nguồn gốc

Khái niệm RAG lần đầu tiên được đề xuất chính thức trong bài báo khoa học "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" bởi Patrick Lewis và các cộng sự tại Facebook AI Research (FAIR) vào năm 2020[a].



Hình 2: Tổng quan kiến trúc RAG trong bài báo gốc của Patrick Lewis (2020).

Trong công trình này, nhóm tác giả định nghĩa RAG là một mô hình xác suất lai (hybrid probabilistic model) kết hợp giữa hai dạng bộ nhớ nhằm khắc phục nhược điểm của các mô hình Pre-trained Seq2Seq truyền thống:

- **Parametric Memory (Bộ nhớ tham số):** Là tri thức ẩn được lưu trữ trong trọng số của một mô hình sinh chuỗi (Pre-trained Seq2Seq Transformer). Cụ thể trong bài báo, tác giả sử dụng mô hình BART (Bidirectional and Auto-Regressive Transformers) làm Generator.
- **Non-Parametric Memory (Bộ nhớ phi tham số):** Là tri thức tường minh từ bên ngoài, cụ thể là một dense vector index chứa các đoạn văn bản Wikipedia. Thành phần này được truy cập thông qua một bộ Neural Retriever dựa trên kiến trúc Dense Passage Retriever.

Cơ chế hoạt động của RAG gốc cho phép Generator (BART) sử dụng đầu vào kết hợp với các tài liệu ẩn (latent documents) tìm được từ Retriever để sinh văn bản. Điểm đặc biệt là toàn bộ kiến trúc này được fine-tuning end-to-end, cho phép cập nhật trọng số của cả Query Encoder và Generator để tối ưu hóa tác vụ đích.

[a]Xem chi tiết tại: https://arxiv.org/abs/2005.11401

II.1. Sự chuyển dịch sang In-Context RAG

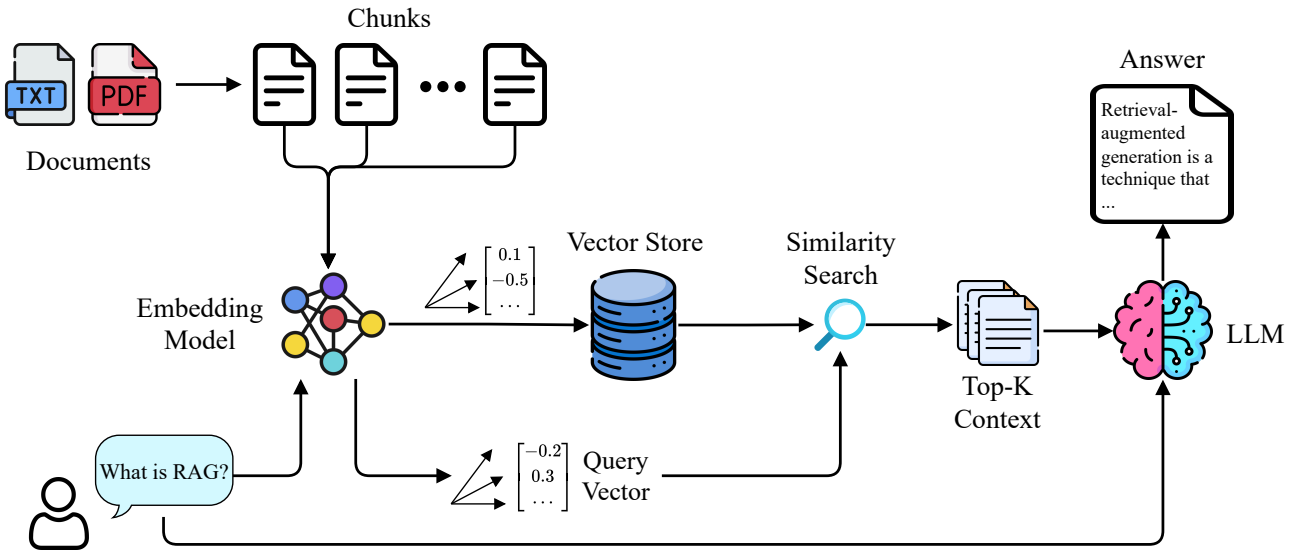
Mặc dù thuật ngữ RAG vẫn được giữ nguyên, tư duy triển khai kỹ thuật này đã thay đổi căn bản cùng với sự phát triển của các LLM:

- **Original RAG (2020):** Là hướng tiếp cận dựa trên fine-tuning. Như đã đề cập ở phần trên, mô hình gốc yêu cầu huấn luyện đồng thời cả bộ truy xuất và mô hình sinh văn bản để chúng học cách phối hợp với nhau. Trọng số của mô hình thay đổi trong quá trình này.
- **Modern RAG (Hiện nay):** Là hướng tiếp cận dựa trên In-Context Learning. Với sự bùng nổ của các LLM cực lớn có khả năng hiểu ngữ cảnh rộng, RAG hiện đại thường ám chỉ quy trình "Retrieve and Prompt".

Trong mô hình hiện đại, chúng ta thường giữ nguyên trọng số của LLM và chỉ tập trung tối ưu hóa việc truy xuất dữ liệu, sau đó đưa dữ liệu này vào đầu vào (Prompt) để mô hình xử lý. Cách tiếp cận này linh hoạt, chi phí thấp và dễ dàng áp dụng cho dữ liệu riêng tư mà không cần quy trình huấn luyện phức tạp.

II.2. Kiến trúc RAG hiện đại

Một hệ thống RAG tiêu chuẩn hiện nay thường được mô hình hóa thành một quy trình gồm 3 giai đoạn chính: Indexing, Retrieval và Generation.



Hình 3: Sơ đồ luồng hoạt động cơ bản của RAG.

II.2.1. Phase 1: Indexing

Giai đoạn này tương đồng với quy trình ETL (Extract-Transform-Load) trong kỹ thuật dữ liệu. Mục tiêu của giai đoạn này là chuyển đổi dữ liệu thô từ nhiều định dạng khác nhau thành một định dạng thống nhất để hệ thống có thể tìm kiếm được.

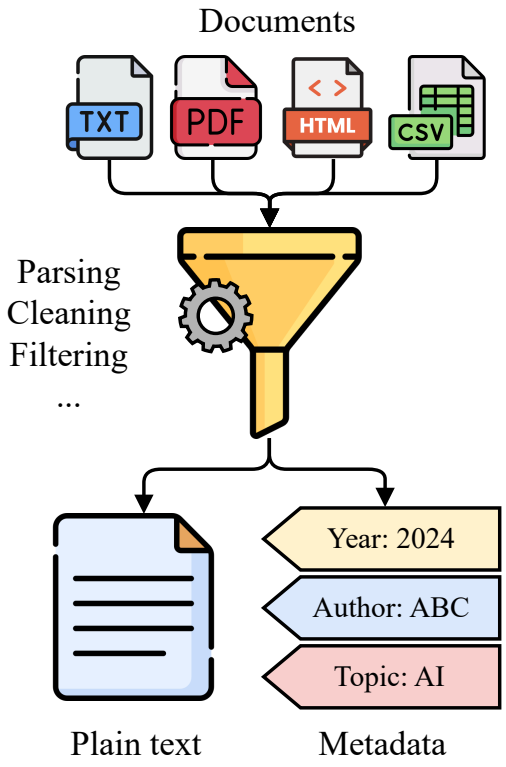
Quy trình này bao gồm các bước từ cơ bản đến nâng cao như sau:

1. **Document Loading**:

   Bắt đầu với việc thu thập nguồn dữ liệu đầu vào, có thể là dữ liệu đã có sẵn trong nội bộ doanh nghiệp hoặc dữ liệu được cào từ internet.

   - **Trích xuất nội dung**: Hệ thống cần có khả năng xử lý đa dạng các loại tệp tin để có thể loại bỏ các định dạng hiển thị phức tạp như font chữ, màu sắc, layout và giữ lại phần quan trọng nhất là văn bản thuần túy.
   - **Thu thập siêu dữ liệu**: Trong các ứng dụng thực tế, chỉ lấy nội dung văn bản là chưa đủ. Hệ thống RAG tối ưu cần trích xuất kèm cả các thông tin ngữ cảnh đi kèm tài liệu, hay còn gọi là metadata, ví dụ: chủ đề, số trang, ngày xuất bản, tác giả, ...

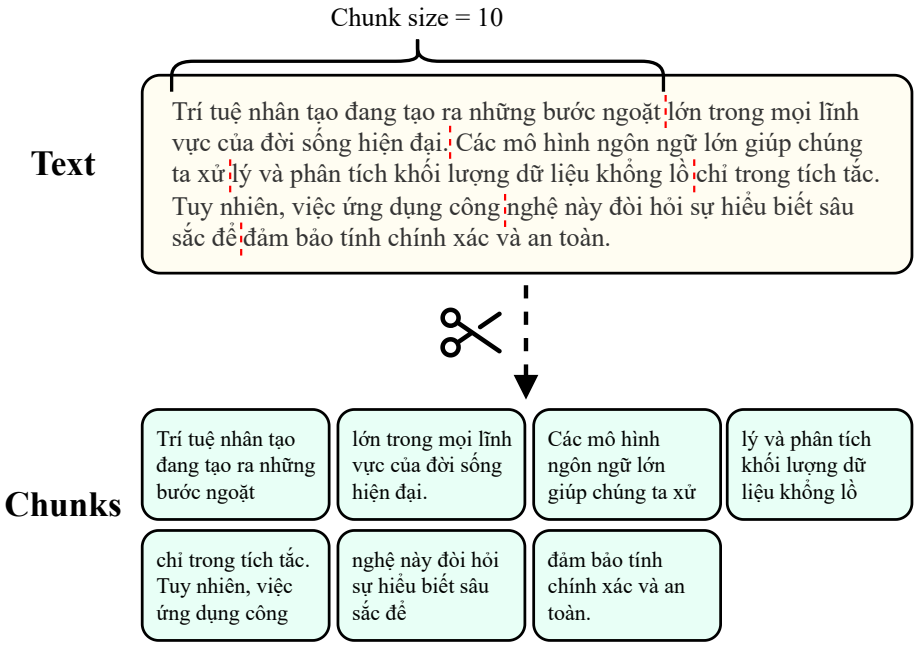
     Vai trò của metadata là cực kỳ quan trọng để hỗ trợ tính năng Pre-filtering. Ví dụ như nếu người dùng hỏi về "Doanh thu năm 2024", hệ thống sẽ dùng metadata để lọc đúng các tài liệu trong năm 2024 thay vì tìm kiếm toàn bộ dữ liệu.



Hình 4: Minh họa quy trình rút trích văn bản từ các nguồn dữ liệu.

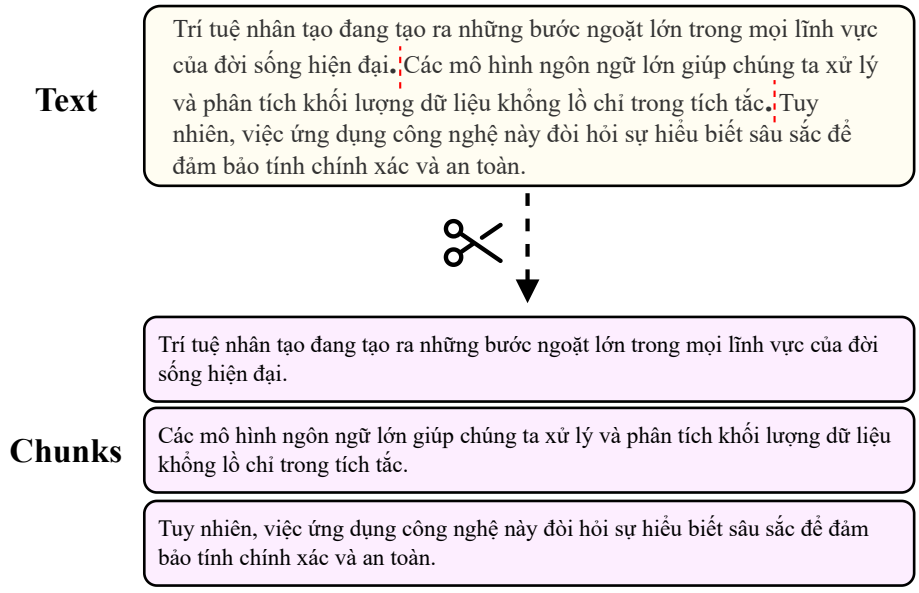
2. **Text Splitting (Chunking)**: Đây là bước chia nhỏ tài liệu dài thành các đoạn nhỏ hơn gọi là "chunks".

   - **Tại sao cần Chunking?**
     (a) Giới hạn Context Window: Các mô hình LLM và Embedding đều có giới hạn về số lượng token đầu vào. Không thể đưa toàn bộ nội dung vào mô hình cùng lúc.
     (b) Độ chính xác của tìm kiếm: Vector của một đoạn văn ngắn tập trung vào một ý tưởng cụ thể sẽ đại diện cho ngữ nghĩa tốt hơn so với vector trung bình cộng của cả một trang giấy chứa nhiều chủ đề hỗn tạp.
   - **Fixed-size Chunking:** Chia văn bản dựa trên số lượng ký tự hoặc token cố định (ví dụ: cứ 500 ký tự thì cắt một lần). Cách này đơn giản nhưng dễ làm mất ngữ nghĩa nếu điểm cắt rơi vào giữa câu hoặc giữa một ý đang trình bày dở.



Hình 5: Minh họa Fixed-size Chunking với Chunk size = 10.

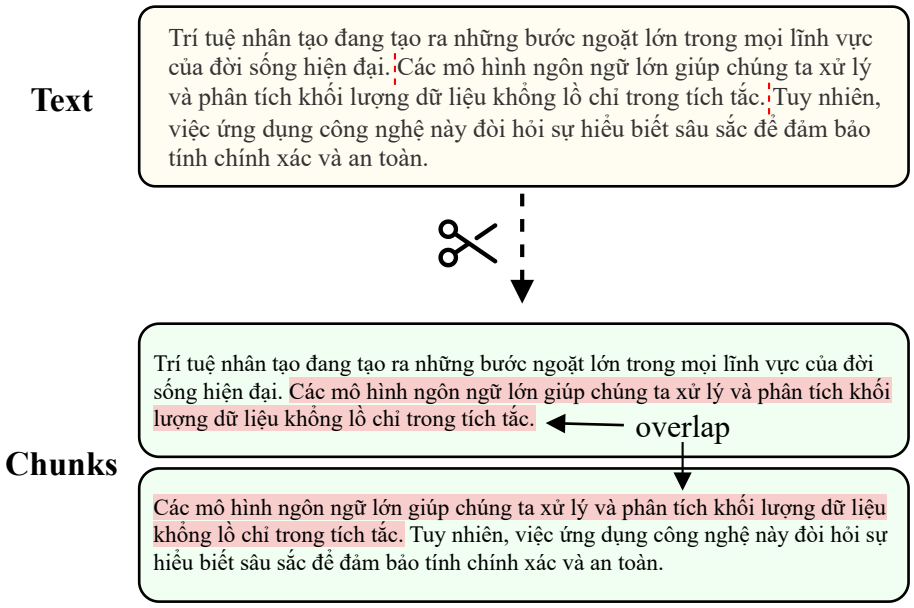
- **Recursive Chunking:** Đây là phương pháp phổ biến nhất hiện nay. Hệ thống sẽ cố gắng cắt dựa trên cấu trúc tự nhiên của văn bản theo thứ tự ưu tiên: dấu ngắt đoạn văn ($\backslash n\backslash n$) → dấu xuống dòng ($\backslash n$) → dấu chấm câu → khoảng trắng. Cách này giúp giữ trọn vẹn cấu trúc câu và đoạn văn.



Hình 6: Minh họa Recursive Chunking bằng dấu chấm câu.

- **Chunk Overlap:** Để đảm bảo ngữ nghĩa không bị mất tại điểm cắt giữa hai chunk liền kề, ta thiết lập tham số `chunk_overlap` (thường là 10-20% độ dài của chunk).

Ví dụ: Chunk 1 kết thúc ở từ thứ 100, thì Chunk 2 sẽ bắt đầu từ từ thứ 80. Phần giao thoa này đóng vai trò như "cầu nối" ngữ cảnh.



Hình 7: Minh họa cho khái niệm Chunk Overlap.

3. **Chiến lược Indexing nâng cao**

   Ở RAG cơ bản, đoạn văn bản dùng để tìm kiếm và đoạn văn bản đưa cho LLM cùng là một. Tuy nhiên, điều này có thể nảy sinh mâu thuẫn: chunk nhỏ tốt cho tìm kiếm, chunk lớn tốt cho LLM do có nhiều ngữ cảnh hơn. Để giải quyết, ta có các chiến lược mở rộng:

   - **Parent-Child Indexing (Small-to-Big):**
     - Chia văn bản thành các khối lớn, ví dụ 1000 tokens, gọi là Parent chunks để chứa ngữ cảnh đầy đủ.
     - Chia Parent chunks thành các khối nhỏ hơn, ví dụ 200 tokens, gọi là Child chunks.
     - **Cơ chế:** Hệ thống sẽ index và tìm kiếm trên Child chunks để đạt độ chính xác cao, nhưng khi trả về kết quả cho LLM, hệ thống sẽ lấy Parent chunk tương ứng.
   - **Summary Indexing:** Sử dụng LLM để tóm tắt chunk gốc. Ta sẽ index phiên bản tóm tắt này với ý nghĩa được cô đọng hơn, nhưng khi cần sẽ trả về văn bản gốc chi tiết.

4. **Embedding**

   - Sử dụng một mô hình Embedding để chuyển đổi các chunks văn bản thành các vector số học dạng dense vector trong không gian nhiều chiều.
   - Theo đó, các đoạn văn bản có nội dung ngữ nghĩa giống nhau sẽ có vector nằm gần nhau trong không gian này.

5. **Vector Store**

   - Lưu trữ các vector cùng với ID và metadata tương ứng vào cơ sở dữ liệu vector chuyên dụng để phục vụ việc truy xuất ở giai đoạn sau.

II.2.2. Phase 2: Retrieval

Đây là thành phần quyết định sự thành bại của hệ thống RAG. Nếu bước này trích xuất thông tin sai hoặc thiếu, LLM sẽ không có đủ dữ liệu để trả lời. Quá trình này không chỉ đơn thuần là tìm kiếm vector mà còn bao gồm các kỹ thuật tối ưu hóa sau:

1. **Query Processing**

   - Ở bước này, hệ thống nhận câu hỏi từ người dùng và đưa qua mô hình Embedding để tạo ra vector truy vấn q.
   - Trong thực tế, câu hỏi của người dùng thường ngắn, thiếu ngữ cảnh hoặc có thể mang nhiều nghĩa. Để khắc phục, ta có thể áp dụng:
     - Multi-Query: Sử dụng LLM để sinh ra 3-5 biến thể khác nhau của câu hỏi gốc, sau đó tìm kiếm tất cả và gộp kết quả lại. Điều này giúp tăng khả năng tìm thấy tài liệu liên quan mà không phụ thuộc vào cách diễn đạt duy nhất của người dùng.

       Ví dụ:

       Input (User): "Lỗi kết nối db" *(Câu hỏi ngắn, thiếu ngữ cảnh cụ thể)*

       LLM Generated Queries:

       (a) "Cách khắc phục lỗi connection timeout khi kết nối database."
       (b) "Xử lý lỗi Access Denied cho user root trong MySQL/PostgreSQL."
       (c) "Hướng dẫn kiểm tra firewall chặn port 5432 hoặc 3306."

       → *Hệ thống sẽ tìm kiếm cả 3 vấn đề (Timeout, Permission, Network) để đảm bảo không bỏ sót tài liệu kỹ thuật liên quan.*

     - HyDE (Hypothetical Document Embeddings): Yêu cầu LLM viết một câu trả lời giả định cho câu hỏi, sau đó dùng vector embedding của câu trả lời giả định này để tìm kiếm. Cách này giúp thu hẹp khoảng cách ngữ nghĩa giữa "câu hỏi" và "tài liệu chứa câu trả lời".

       Ví dụ:

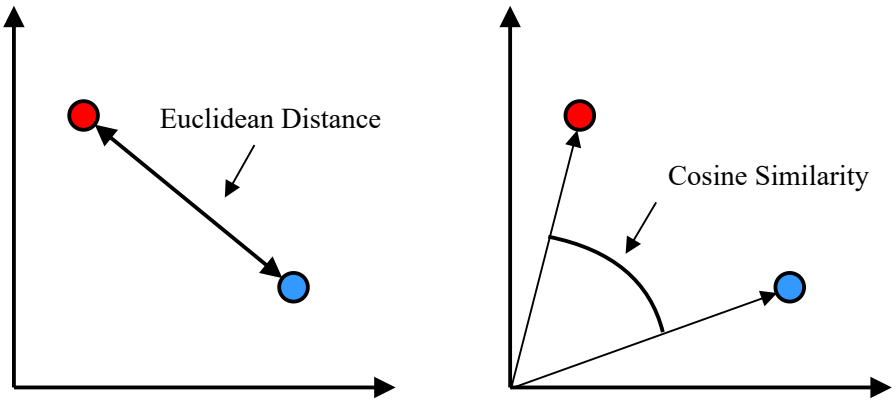
       Input (User): "Quy định mang laptop ra ngoài"

       Hypothetical Document (LLM giả định): *"Theo chính sách bảo mật thông tin, nhân viên muốn mang thiết bị tài sản công ty (laptop, tablet) ra khỏi văn phòng cần điền phiếu 'Đăng ký mang thiết bị' trên hệ thống và phải được Trưởng bộ phận phê duyệt..."*

       → *Vector của đoạn văn giả định này sẽ khớp tốt với tài liệu "Sổ tay nhân viên" hoặc "Quy trình an ninh" trong database, tốt hơn nhiều so với câu hỏi gốc.*

2. **Similarity Search**

- Dense Retrieval: Hệ thống tính toán độ tương đồng giữa vector truy vấn q và các vector tài liệu d trong cơ sở dữ liệu.



Hình 8: Minh họa hai metric đo độ tương đồng hoặc khoảng cách thường được dùng.

  - Metric: Sử dụng Cosine Similarity hoặc Euclidean Distance.
  - Algorithm: Để đảm bảo tốc độ khi dữ liệu lớn, thay vì so khớp từng đôi một (thường được gọi là Brute-force), ta sử dụng thuật toán ANN (Approximate Nearest Neighbor) như HNSW để tìm kiếm lân cận gần đúng với độ trễ thấp.
- **Vấn đề:** Tìm kiếm vector rất tốt về ngữ nghĩa nhưng lại yếu về từ khóa chính xác. Ví dụ: vector của "iPhone 14" và "iPhone 15" rất gần nhau về ngữ nghĩa, nhưng người dùng cần tìm chính xác thông tin về phiên bản 15.

3. **Hybrid Search**

Đây là kỹ thuật mở rộng tiêu chuẩn để khắc phục nhược điểm trên.

- **Cơ chế**: Kết hợp kết quả từ hai luồng tìm kiếm song song:
  - Dense Retrieval (Vector Search): Tìm kiếm dựa trên độ tương đồng ngữ nghĩa trong không gian vector. Phương pháp này giỏi việc tìm kiếm các nội dung có ý nghĩa tương đương (ví dụ: "giá xe" $\leftrightarrow$ "chi phí mua ô tô").
  - Sparse Retrieval (Keyword Search): Sử dụng các thuật toán thống kê truyền thống như BM25 hoặc TF-IDF.
    $\rightarrow$ *Giải thích:* Các thuật toán này hoạt động bằng cách đếm tần suất xuất hiện chính xác của từ khóa. Nó cực kỳ quan trọng để bắt chính xác các tên riêng, từ khóa hoặc thuật ngữ chuyên ngành mà Vector Search thường bỏ qua.
- **Reciprocal Rank Fusion (RRF):** Là thuật toán hậu xử lý dùng để gộp và xếp hạng lại kết quả từ hai luồng trên.

Cách hoạt động của thuật toán RRF

Vấn đề: Điểm số của Vector Search và BM25 có thang đo hoàn toàn khác nhau, việc cộng trực tiếp sẽ không hợp lí.

Giải pháp: RRF bỏ qua điểm số gốc và chuẩn hóa dựa trên **thứ hạng**.

$$Score = \sum_i \frac{1}{k + r_i}$$

- Cơ chế: Sử dụng nghịch đảo của thứ hạng (r_i). Hạng 1 (mẫu số nhỏ) sẽ cho điểm rất lớn, Hạng 100 (mẫu số lớn) sẽ cho điểm rất nhỏ.
- Hằng số k: Thường là 60, giúp làm mượt điểm số, ngăn không cho các tài liệu Top 1 chiếm ưu thế quá tuyệt đối so với Top 2, Top 3.
- Kết quả: Tài liệu đứng hạng trung bình ở cả 2 danh sách thường có tổng điểm cao hơn tài liệu chỉ đứng nhất ở 1 danh sách nhưng mất tích ở danh sách kia.

4. **Re-ranking**

Sau khi có được tập ứng viên (ví dụ: Top 50 tài liệu) từ bước trên, thứ tự của chúng có thể chưa hoàn toàn chính xác do vector chỉ là dạng nén thông tin.

- **Cross-Encoder**: Sử dụng một mô hình Deep Learning chuyên biệt để chấm điểm lại mức độ liên quan giữa câu hỏi và từng tài liệu trong tập ứng viên ban đầu.
- **Tại sao cần Cross-Encoder?**
  - Bi-Encoder (Dùng ở bước Indexing/Retrieval): Mã hóa câu hỏi và tài liệu thành 2 vector riêng biệt một cách độc lập. Ưu điểm là cực nhanh, nhưng nhược điểm là làm mất mối quan hệ ngữ pháp và ngữ nghĩa phức tạp giữa câu hỏi và văn bản.
  - Cross-Encoder (Dùng ở bước Re-ranking này): Đưa cả câu hỏi và văn bản vào mô hình cùng một lúc (như con người đọc song song). Nó có thể nhận biết các sắc thái phủ định, quan hệ nguyên nhân-kết quả phức tạp. Chính xác hơn nhưng chậm hơn nhiều.
- **Chiến lược "Hình phễu"**: Retrieve Many (Lấy 50 tài liệu nhanh bằng Bi-Encoder) → Re-rank Few (Lấy 5 tài liệu tốt nhất bằng Cross-Encoder) → Đưa vào LLM. Cách này cân bằng giữa tốc độ và độ chính xác.

Ví dụ:

Query (User): "Tại sao tôi không nhận được thông báo qua email?"

1. Kết quả từ Vector Search (Bi-Encoder): *(Tìm dựa trên độ tương đồng chung về chủ đề "email" và "thông báo")*

A. Rank 1: "Hướng dẫn cài đặt chữ ký email." *(Sai: cùng chủ đề nhưng sai ý định)*
B. Rank 2: "Quy định về văn hóa gửi email công ty." *(Sai)*
C. Rank 3: "Xử lý lỗi email bị rơi vào thư mục Spam." *(Đúng, nhưng bị xếp hạng thấp)*

2. Sau khi qua Cross-Encoder (Re-ranking): *(Mô hình đọc hiểu mối quan hệ "không nhận được" → "lỗi/spam")*

A. Rank 1: "Xử lý lỗi email bị rơi vào thư mục Spam." ↑ (Được đẩy lên đầu)
B. Rank 2: "Hướng dẫn cài đặt chữ ký email."
C. Rank 3: "Quy định về văn hóa gửi email công ty."

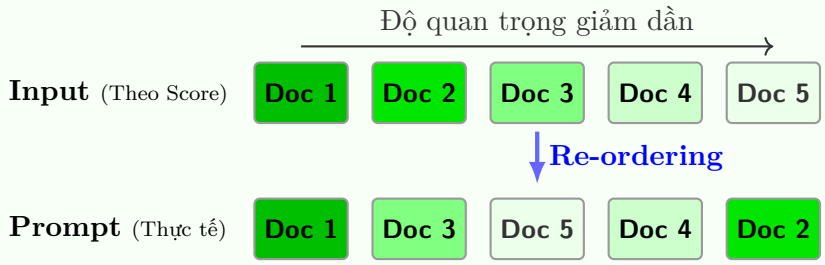
II.2.3. Phase 3: Generation

Sau khi Retrieval trả về danh sách các tài liệu liên quan, hệ thống chuyển sang giai đoạn sinh văn bản. Đây là lúc LLM tổng hợp thông tin để trả lời người dùng. Quá trình này không chỉ đơn giản là đưa dữ liệu vào, mà còn đòi hỏi các chiến lược xử lý ngữ cảnh tinh tế.

1. **Context Preparation**
   - Context Stuffing: Đây là phương pháp đơn giản nhất. Hệ thống gộp toàn bộ văn bản của các tài liệu tìm được (Top-K) thành một đoạn văn dài, sau đó ghép vào Prompt.
   - **Vấn đề**:
     - Chi phí & Độ trễ: Input càng dài, LLM xử lý càng chậm và tốn token.
     - Nhiễu thông tin: Quá nhiều thông tin không liên quan có thể khiến LLM bị "loãng" chủ đề.
   - **Context Selection & Compression**:
     - Context Reordering: Dựa trên hiện tượng "Lost in the Middle": LLM thường chú ý tốt nhất vào thông tin ở đầu và cuối prompt, nhưng hay bỏ quên thông tin ở giữa.

Chiến thuật Reordering: Tối ưu hóa vị trí theo dạng hình chữ U

Thay vì đưa tài liệu vào theo thứ tự độ liên quan giảm dần, ta sắp xếp lại để đưa các tài liệu quan trọng nhất nằm ở hai đầu và các tài liệu ít quan trọng vào giữa:



→ **Kết quả:** Các tài liệu quan trọng (màu đậm) bao bọc lấy các tài liệu ít quan trọng hơn (màu nhạt), tận dụng tối đa khả năng tập trung của LLM.

   - Context Compression: Sử dụng một LLM nhỏ hoặc thuật toán NLP để tóm tắt ý chính trước khi đưa vào LLM chính.

Ví dụ:

Query: "Phí hủy dịch vụ là bao nhiêu?"

1. Raw Context (300 tokens): *"...Theo điều khoản 7.2 của hợp đồng dịch vụ được ký kết vào ngày... (văn bản pháp lý dài dòng)... trong trường hợp khách hàng muốn chấm dứt hợp đồng trước thời hạn, cần thông báo bằng văn bản trước 30 ngày và chịu khoản phí phạt tương đương 02 tháng cước sử dụng..."*

2. Compressed Context (20 tokens): *"Điều khoản 7.2: Phí hủy dịch vụ là 02 tháng cước sử dụng."*

→ *LLM chính chỉ cần nhận đoạn ngắn gọn này, giúp trả lời nhanh hơn và giảm thiểu hallucination.*

2. **Prompt Engineering**

Chất lượng câu trả lời phụ thuộc rất lớn vào cách ta ra lệnh cho mô hình. Một context tốt nhưng prompt tệ vẫn có thể dẫn đến câu trả lời sai.

- **Zero-shot Prompting:** Sử dụng một template cố định để hướng dẫn mô hình trả lời trực tiếp dựa trên ngữ cảnh mà không cần ví dụ mẫu.

RAG Prompt Template (Standard)

```
System: You are an assistant for question-answering tasks. Use the
following pieces of context to answer the question. If you don't know
the answer, just say that you don't know.

Context: {context}
Question: {question}
Answer:
```

- **Few-shot Learning:** Cung cấp thêm 1-2 ví dụ mẫu (gồm bộ ba Context - Question - Answer chuẩn) vào prompt để LLM học được phong cách trả lời và định dạng mong muốn.

Ví dụ

Instruction: Trả lời ngắn gọn, chỉ đưa ra con số.

Example 1:
Context: "Doanh thu năm 2022 là 10 tỷ, năm 2023 tăng lên 12 tỷ."
Question: Doanh thu 2023 bao nhiêu?
Answer: 12 tỷ.

Actual Task:
Context: {real_context}
Question: {user_question}
Answer:

- **Chain-of-Thought (CoT - Chuỗi suy luận):** Yêu cầu mô hình suy nghĩ từng bước (*Let's think step by step*) dựa trên các dữ kiện trong context trước khi đưa ra kết luận cuối cùng.

Tại sao cần CoT?

Đối với các câu hỏi phức tạp đòi hỏi logic, nếu ép LLM trả lời ngay lập tức, nó rất dễ đoán mò.

Với CoT: LLM tự sinh ra quy trình: *"Bước 1: Tìm giá gói A... Bước 2: Tìm giá gói B... Bước 3: So sánh..."* → Kết luận chính xác hơn đáng kể[a].

[a] Xem chi tiết tại: https://arxiv.org/abs/2201.11903

3. **Generation & Attribution**

- **Cơ bản:** LLM sinh ra câu trả lời dưới dạng văn bản thông thường. Mục tiêu chính là tính trôi chảy và đúng ngữ pháp.
- **Citation:** Một trong những lợi thế cạnh tranh lớn nhất của RAG so với các chatbot truyền thống là khả năng minh bạch hóa nguồn tin.
  - Cơ chế: Trong prompt, ta yêu cầu LLM: "Mọi thông tin đưa ra phải kèm theo ID của tài liệu nguồn".
  - Lợi ích: Giúp người dùng dễ dàng kiểm chứng, xây dựng niềm tin và giảm thiểu rủi ro khi mô hình bịa đặt thông tin.

Minh họa output có trích dẫn nguồn

User Question: "Quy định làm thêm giờ tính thế nào?"

RAG Response: "Theo quy định công ty, nhân viên làm thêm giờ vào ngày thường được hưởng 150% lương cơ bản **[Sổ tay nhân viên, Tr.12]**. Đối với ngày lễ tết, mức hưởng là 300% lương **[Luật lao động 2019, Điều 98]**."

→ *Người dùng có thể click vào [..] để mở trực tiếp tài liệu gốc đối chiếu.*

II.3. Mở rộng: Phân tích vai trò của các thành phần

Để đánh giá chính xác tầm quan trọng của từng module trong kiến trúc RAG, chúng ta cùng phân tích các giả định theo câu hỏi: "Hệ thống sẽ hoạt động như thế nào nếu loại bỏ một thành phần cụ thể?".

Kịch bản 1: Loại bỏ Embedding Model

Vai trò Embedding Model: Chuyển đổi văn bản sang vector để hỗ trợ truy vấn theo ngữ nghĩa.

- **Cơ chế hoạt động:** Khi không có Embedding Model, hệ thống không thể mã hóa văn bản thành vector và mất khả năng so sánh tương đồng ngữ nghĩa. Buộc phải quay về các phương pháp truy vấn theo mặt chữ (Lexical Retrieval) như BM25, TF-IDF hoặc Exact Match.
- **Tác động:** Giảm mạnh khả năng bắt ngữ cảnh, từ đồng nghĩa và cách diễn đạt tương đương.
  Ví dụ: Người dùng tìm "giá xe hơi". Tài liệu chỉ chứa "chi phí mua ô tô" có thể bị bỏ qua vì không khớp từ khóa trực tiếp.
- **Hệ quả:** Hệ thống trở thành Lexical Retrieval-driven RAG: Vẫn có LLM để trả lời, nhưng chất lượng truy vấn theo ngữ nghĩa giảm rõ rệt và dễ bỏ sót thông tin.

Kịch bản 2: Loại bỏ Vector Store/ANN Index

Vai trò Vector Store/ANN Index: Lưu trữ vector, quản lý metadata và hỗ trợ tìm kiếm nhanh ở quy mô lớn.

- **Cơ chế hoạt động:** Hệ thống vẫn có thể tạo vector, nhưng không có lớp chỉ mục ANN, việc tìm kiếm thường phải so sánh với nhiều vector theo kiểu tuyến tính (Brute-force).
- **Tác động:** Độ trễ truy vấn tăng mạnh khi dữ liệu lớn ($O(N)$). Với vài trăm tài liệu vẫn chấp nhận được, nhưng với hàng trăm nghìn đến hàng triệu vector, hệ thống dễ chậm, tốn tài nguyên và khó đáp ứng thời gian thực.
  Ngoài ra, việc lọc theo metadata, cập nhật/chèn/xóa dữ liệu, và vận hành cũng trở nên khó khăn nếu không có lớp lưu trữ phù hợp.
- **Hệ quả:** Hệ thống trở thành Unscalable Prototype, có thể chạy demo ở quy mô nhỏ, nhưng khó mở rộng và khó vận hành ổn định trong môi trường thực tế.

Kịch bản 3: Loại bỏ LLM (Large Language Model)

Vai trò LLM: Bộ não Tổng hợp thông tin, suy luận và sinh câu trả lời dựa trên ngữ cảnh truy vấn.

- **Cơ chế hoạt động:** Hệ thống vẫn có thể truy vấn tốt các tài liệu liên quan nhờ Embedding Model và Vector Store, nhưng quy trình dừng ở bước Retrieval. Output trả về chỉ là danh sách các đoạn văn bản (Top-K) kèm điểm số/metadata.
- **Tác động:** Mất khả năng tổng hợp, diễn giải và trả lời trực tiếp theo dạng hội thoại (Question Answering).
  Ví dụ: Thay vì trả lời "Doanh thu là 10 tỷ", hệ thống chỉ trả về các đoạn trích từ báo cáo, và người dùng phải tự đọc để rút kết luận.
- **Hệ quả:** Hệ thống trở thành Semantic Search / Retrieval System, mạnh về truy vấn theo ngữ nghĩa, nhưng không còn thành phần Generation, nên chưa phải là chatbot hay hệ RAG hoàn chỉnh.

Thành phần thiếu	Hệ thống trở thành	Hạn chế cốt lõi
Embedding Model	**Lexical Retrieval-driven RAG** RAG dựa trên truy vấn từ khóa	Giảm mạnh truy vấn theo ngữ nghĩa, khó bắt từ đồng nghĩa, và dễ bỏ sót thông tin.
Vector Store/ANN Index	**Unscalable Prototype** Mô hình khó mở rộng	Độ trễ tăng mạnh ở dữ liệu lớn, khó vận hành hệ thống.
LLM	**Semantic Search / Retrieval System** Hệ truy vấn ngữ nghĩa	Không thể tổng hợp, diễn giải, và trả lời hội thoại; người dùng phải tự đọc và rút kết luận từ các đoạn trích.

Bảng 1: Tổng hợp sự thay đổi của hệ thống RAG theo các kịch bản.

III. Thư viện LangChain

III.1. Giới thiệu

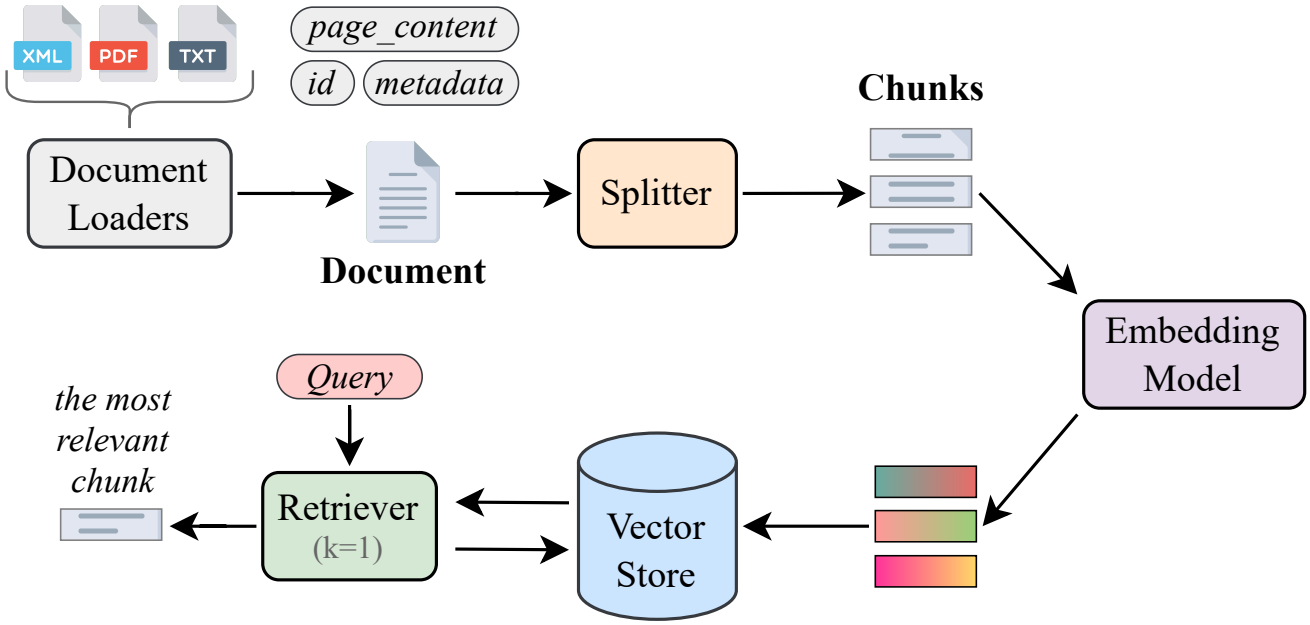
LangChain là một framework mã nguồn mở mạnh mẽ, được thiết kế để đơn giản hóa quá trình phát triển các ứng dụng sử dụng mô hình ngôn ngữ lớn (LLM). Một cách tổng quát, LangChain đóng vai trò liên kết các thành phần dữ liệu, xử lý logic và mô hình lại với nhau.

Trong bài toán Retrieval Augmented Generation (RAG), LangChain giải quyết vấn đề cốt lõi là sự rời rạc giữa nguồn dữ liệu và khả năng của LLM. Thư viện này hỗ trợ toàn bộ vòng đời của

ứng dụng RAG mà không cần người lập trình phải tự xây dựng các kết nối thủ công phức tạp.

III.2. Các thành phần cốt lõi

Để xây dựng một ứng dụng RAG với LangChain, ta cần phải hiểu về các thành phần chính như bên dưới:



Hình 9: Tổng quan các thành phần chính của RAG trong LangChain.

1. **Documents và Document Loaders**

   Trong LangChain, đơn vị cơ bản để biểu diễn thông tin là đối tượng `Document`. Trước khi xây dựng RAG, dữ liệu từ các nguồn khác nhau cần được chuyển đổi về định dạng này.

   Cấu trúc Document: LangChain triển khai lớp `Document` để đại diện cho một đơn vị văn bản và các siêu dữ liệu đi kèm, thường được gọi là metadata. Một đối tượng `Document` bao gồm ba thuộc tính chính:

   - `page_content`: Chuỗi ký tự chứa nội dung văn bản.
   - `metadata`: Một dictionary chứa thông tin bổ sung tùy ý, ví dụ như nguồn gốc của tài liệu, số trang, quan hệ với tài liệu khác.
   - `id`: Mã định danh chuỗi cho tài liệu.

   Loading Documents: LangChain cung cấp hệ sinh thái **Document Loaders** tích hợp với hàng trăm nguồn dữ liệu khác nhau như PDF, CSV, HTML. Ví dụ, sử dụng `PyPDFLoader` để nạp một tài liệu PDF thành một `Document`.

```
from langchain_community.document_loaders import PyPDFLoader

```

```
file_path = "../ai-vietnam-2025.pdf"
loader = PyPDFLoader(file_path)
doc = loader.load()
```

Splitting: Một trang tài liệu gốc thường quá dài hoặc chứa nhiều thông tin hỗn tạp. Việc chia nhỏ văn bản giúp tăng độ chính xác khi truy xuất. Công cụ **Text Splitters** được sử dụng để chia tài liệu thành các đoạn nhỏ, các đoạn này được gọi là các "chunk". Phương pháp phổ biến là `RecursiveCharacterTextSplitter`, với chức năng:

- Phân chia đệ quy dựa trên các ký tự phân cách như xuống dòng, khoảng trắng, để giữ nguyên vẹn ngữ nghĩa.
- Giữ lại một phần ký tự trùng lặp giữa các đoạn liền kề để đảm bảo ngữ cảnh không bị ngắt quãng.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # Kích thước của mỗi chunk
    chunk_overlap=200,  # Giữ 200 tokens của đoạn liền kề
)
all_splits = text_splitter.split_documents(doc)
```

2. **Embeddings**

Vector Search là phương pháp phổ biến để lưu trữ và tìm kiếm trên dữ liệu phi cấu trúc.

Nguyên lý hoạt động: Mô hình Embedding chuyển đổi một văn bản thành một vector số thực biểu diễn cho văn bản ấy. Việc biểu diễn dưới dạng số học cho phép:

- Những văn bản có ý nghĩa tương tự sẽ có vector gần nhau trong không gian hình học.
- Các chỉ số đo lường như cosine similarity được dùng để xác định độ tương đồng giữa các văn bản thông qua tính toán trên vector tương ứng của chúng.

LangChain hỗ trợ giao diện chuẩn cho nhiều nhà cung cấp mô hình embedding như OpenAI, Google, HuggingFace, ...

```
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")

# Tạo vector cho một đoạn văn bản
vector_1 = embeddings.embed_query(all_splits[0].page_content)
print(len(vector_1))
# Kết quả: 1536 (số chiều của vector)
```

3. **Vector Stores**

   `VectorStore` trong LangChain chịu trách nhiệm lưu trữ các đối tượng `Document` và vector tương ứng, đồng thời cung cấp phương thức để truy vấn tới chúng. Gồm hai chức năng chính:

   - **Indexing**: Thêm văn bản vào kho lưu trữ thông qua phương thức `add_documents`.
   - **Querying**: Tìm kiếm văn bản dựa trên độ tương đồng giữa hai vector với phương thức `similarity_search`.

   LangChain tích hợp với nhiều loại vector store: từ dạng lưu trong bộ nhớ như `FAISS` và `InMemoryVectorStore`, đến các cơ sở dữ liệu chuyên dụng với `Chroma`, `Pinecone`, `Postgres`.

```python
from langchain_core.vectorstores import InMemoryVectorStore

vector_store = InMemoryVectorStore(embeddings)
ids = vector_store.add_documents(documents=all_splits)
results = vector_store.similarity_search(
    "How many distribution centers does Nike have in the US?"
)
```

4. **Retrievers**

   Trong khi `VectorStore` là nơi lưu trữ, **Retriever** được dùng để truy vấn dữ liệu. Điểm đặc biệt là Retrievers trong LangChain là các `Runnable`, cho phép chúng kết nối dễ dàng vào các chuỗi xử lý hay "chains".

   VectorStoreRetriever

   Cách để tạo Retriever đơn giản nhất là từ một VectorStore thông qua hàm `.as_retriever()`. Trong đó, có thể bổ sung các tham số tìm kiếm như `search_kwargs` và loại tìm kiếm `search_type`:

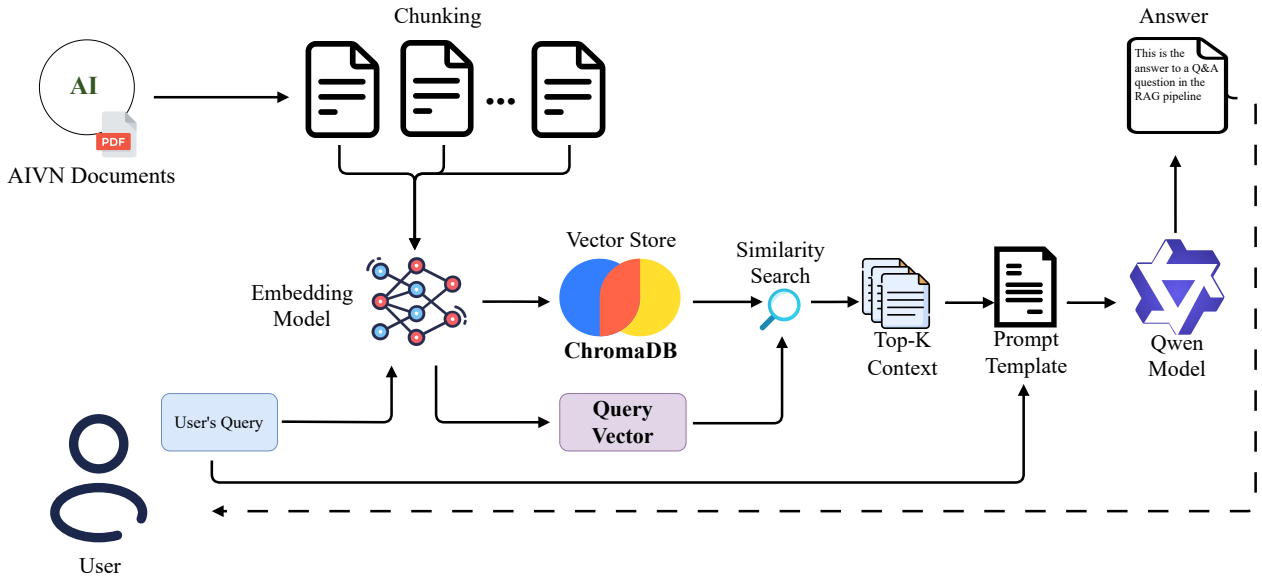
   - `similarity`: Tìm kiếm tương đồng mặc định.
   - `mmr` (Maximum Marginal Relevance): Cân bằng giữa độ tương đồng và độ đa dạng của kết quả.
   - `similarity_score_threshold`: Lọc bỏ các kết quả có điểm tương đồng thấp hơn ngưỡng quy định.

```python
# Tạo Retriever từ Vector Store
retriever = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 1} # Chỉ lấy 1 kết quả tốt nhất
)

# Thực thi tìm kiếm (Batch)
retriever.batch([
    "How many distribution centers does Nike have in the US?",
```

```
    "When was Nike incorporated?"
])
```

IV. Thực hành

Trong phần này, chúng ta sẽ hiện thực hoá các khái niệm về RAG và LangChain bằng việc xây dựng một Question Answering System (tạm dịch: hệ thống hỏi đáp) dành cho các tài liệu PDF tiếng Việt.



Hình 10: Kiến trúc hệ thống RAG sử dụng ChromaDB và mô hình ngôn ngữ Qwen.

Quan sát Hình 10, quy trình phần thực hành bắt đầu từ việc xử lý tài liệu đầu vào, sau đó thực hiện chia nhỏ và vector hoá để lưu trữ vào ChromaDB. Khi người dùng đặt câu hỏi, hệ thống sẽ tìm kiếm ngữ cảnh phù hợp và đưa vào mô hình Qwen để sinh câu trả lời.

Về mặt kỹ thuật, chúng ta sẽ làm việc trên Google Colab với GPU T4, Python phiên bản 3.12, sử dụng nguồn dữ liệu từ các bài viết public của AI VIET NAM. Quy trình triển khai chi tiết sẽ đi qua các bước: thiết lập môi trường làm việc và tổ chức cấu trúc dự án, thu thập và xử lý dữ liệu, xây dựng cơ sở dữ liệu vector, khởi tạo mô hình ngôn ngữ và cuối cùng là phát triển giao diện tương tác.

Các giai đoạn dưới đây được trình bày theo đúng thứ tự của các cell trong notebook, giúp bạn vừa nắm được ý tưởng, vừa dễ dàng chạy thử từng đoạn mã minh họa.

IV.1. Chuẩn bị môi trường và cấu trúc dự án

Ở bước đầu tiên, chúng ta thiết lập môi trường làm việc và tổ chức lại cấu trúc thư mục nhằm đảm bảo toàn bộ mã nguồn vận hành có hệ thống, dễ mở rộng và dễ tích hợp các thành phần của pipeline RAG trong các bước tiếp theo.

```
# Cài đặt các thư viện chính cho mô hình ngôn ngữ, embedding, RAG và giao diện web
!pip install -q \
  "torch>=2.0.0" \
  "transformers>=4.40.0" \
  "accelerate>=0.30.0" \
  "huggingface-hub>=0.23.0" \
  "sentence-transformers>=2.7.0" \
  "langchain>=0.2.0" \
  "langchain-core>=0.2.0" \
  "langchain-community>=0.1.0" \
  "langchain-text-splitters>=0.2.0" \
  "chromadb>=0.5.0" \
  "langchain-chroma>=0.2.0" \
  "pypdf>=4.2.0" \
  "langserve[all]>=0.1.0" \
  "fastapi>=0.115.0" \
  "uvicorn>=0.30.0" \
  "gradio>=5.0.0" \
  "langchain-huggingface" \
  "wget"
```

Đoạn mã sau giúp thiết lập thư mục gốc của dự án trên Colab và tạo các thư mục con để lưu dữ liệu và mã nguồn. Cách tổ chức này giúp chúng ta dễ quản lý khi số lượng file tăng lên.

```
import os
import sys

PROJECT_ROOT = "/content/rag_langchain"

# Đặt token để tải mô hình private, hãy thay bằng token của bạn
os.environ["HF_TOKEN"] = "YOUR_HUGGINGFACE_TOKEN"
# Tạo thư mục cho dữ liệu và mã nguồn
os.makedirs(os.path.join(PROJECT_ROOT, "data_source", "generative_ai"), exist_ok=True)
os.makedirs(os.path.join(PROJECT_ROOT, "src", "base"), exist_ok=True)
os.makedirs(os.path.join(PROJECT_ROOT, "src", "rag"), exist_ok=True)

os.chdir(PROJECT_ROOT)

# Thêm PROJECT_ROOT vào sys.path để có thể import các module trong src
if PROJECT_ROOT not in sys.path:
    sys.path.append(PROJECT_ROOT)

os.environ["TOKENIZERS_PARALLELISM"] = "false"
```

Cuối cùng, chúng ta tạo các tệp `__init__.py` để các thư mục mã nguồn có thể được sử dụng như các package Python, giúp việc import module trở nên rõ ràng hơn.

```
%%bash
touch /content/rag_langchain/src/__init__.py
```

```
touch /content/rag_langchain/src/base/__init__.py
touch /content/rag_langchain/src/rag/__init__.py
```

IV.2. Chuẩn bị dữ liệu PDF đầu vào

Tại bước này, ta tải các tài liệu PDF dùng làm dữ liệu cho hệ thống RAG. Các tài liệu này được trích từ các bài viết công khai của AI VIET NAM, sau đó được lưu vào một thư mục chung để phục vụ cho bước xử lý văn bản và xây dựng vector database.

```
import os
import wget

# Thư mục lưu các tài liệu PDF
DATA_DIR = "/content/rag_langchain/data_source/generative_ai"
os.makedirs(DATA_DIR, exist_ok=True)

# Danh sách các tài liệu PDF mẫu sẽ dùng làm dữ liệu cho hệ thống RAG
pdf_links = [
    {
        "title": "Vòng lặp for và ứng dụng",
        "url": "https://docs.google.com/uc?export=download&id=
                                        1_zJnj5qORwzMH6vgftGzkYu4fFew6Pq4"
    },
    {
        "title": "Giám sát hệ thống AI với Grafana và Prometheus",
        "url": "https://docs.google.com/uc?export=download&id=1gZWLJddiuLd-
                                        ZZ8j_Nmfu_r7fISGRW6J"
    },
    {
        "title": "Các Thước Đo Đánh Giá Mô Hình Hồi Quy",
        "url": "https://docs.google.com/uc?export=download&id=1C-
                                        f9pNW0mkMxaakDcpliN3isTVqRQtR3"
    },
    {
        "title": "A simple, strong baseline for Long-Term Forecasts",
        "url": "https://docs.google.com/uc?export=download&id=
                                        16KFeWi0ONqV3ZJYAgC_20y7bxUNGd9hU"
    },
]

# Tải các file PDF về thư mục dữ liệu nếu chưa tồn tại
for pdf_info in pdf_links:
    save_path = os.path.join(DATA_DIR, f"{pdf_info['title']}.pdf")
    if not os.path.exists(save_path):
        try:
            wget.download(pdf_info["url"], out=save_path)
        except Exception as e:
            pass
```

IV.3. Tiền xử lý văn bản và Chunking

Ở bước này, ta làm sạch văn bản trích xuất từ các file PDF và chia nội dung thành các đoạn nhỏ phù hợp. Việc tiền xử lý giúp văn bản gọn, ít lỗi, còn việc chia đoạn giúp mô hình embedding và mô hình ngôn ngữ làm việc hiệu quả hơn.

Làm sạch văn bản tiếng Việt

Văn bản trích xuất từ PDF thường chứa nhiều ký tự đặc biệt, khoảng trắng thừa, hoặc các vấn đề về encoding. Hàm `clean_vietnamese_text` dưới đây giúp chuẩn hóa văn bản tiếng Việt, loại bỏ các ký tự không mong muốn và đảm bảo encoding đồng nhất.

```python
import re
import unicodedata
from typing import List

def clean_vietnamese_text(text: str) -> str:
    # Chuẩn hóa Unicode về dạng NFC cho tiếng Việt
    text = unicodedata.normalize('NFC', text)
    text = "".join(
        char for char in text
        if not unicodedata.category(char).startswith('C') or char in '\n\t'
    )

    # Gộp khoảng trắng thừa và dòng trống
    text = re.sub(r'\s+', ' ', text)
    text = re.sub(r'\n\s*\n', '\n', text)
    return text.strip()
```

Tải và chia nhỏ tài liệu PDF

Sau khi đã có hàm làm sạch văn bản, ta xây dựng hai class chính: `SimpleLoader` để tải tài liệu PDF và `TextSplitter` để chia văn bản thành các chunk phù hợp với mô hình.

```python
import glob
from tqdm import tqdm
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter

class SimpleLoader:
    # Tải một file PDF và gọi hàm làm sạch nội dung văn bản
    def load_pdf(self, pdf_file: str):
        docs = PyPDFLoader(pdf_file, extract_images=True).load()
        for doc in docs:
            doc.page_content = clean_vietnamese_text(doc.page_content)
        return docs

```

```
    # Tải tất cả file PDF trong một thư mục
    def load_dir(self, dir_path: str) -> List:
        pdf_files = glob.glob(f"{dir_path}/*.pdf")
        if not pdf_files:
            raise ValueError(f"No PDF files found in {dir_path}")

        all_docs = []
        for pdf_file in tqdm(pdf_files, desc="Loading PDFs"):
            try:
                all_docs.extend(self.load_pdf(pdf_file))
            except Exception as e:
                pass
        return all_docs

class TextSplitter:

    # Khởi tạo với chunk_size và chunk_overlap phù hợp cho tiếng Việt
    def __init__(self,
        chunk_size: int = 400,
        chunk_overlap: int = 120,
    ):
        self.splitter = RecursiveCharacterTextSplitter(
            separators=["\n\n", "\n", " ", ""],
            chunk_size=chunk_size,
            chunk_overlap=chunk_overlap,
            length_function=len,
        )

    # Chia danh sách documents thành các chunk nhỏ hơn
    def split(self, documents):
        return self.splitter.split_documents(documents)
```

IV.4. Xây dựng Vector Database

Tại bước này, các chunk đã được chia nhỏ sẽ được chuyển đổi thành vector thông qua mô hình embedding đa ngôn ngữ từ HuggingFace, sau đó lưu trữ vào Chroma. Class `VectorDB` lúc này đóng vai trò trung tâm, quản lý toàn bộ quy trình từ khởi tạo embedding model, index các document chunks, cho đến việc cung cấp retriever phục vụ cho việc tìm kiếm ngữ nghĩa.

```
from langchain_chroma import Chroma
from langchain_huggingface import HuggingFaceEmbeddings

class VectorDB:
    def __init__(
        self,
        documents=None,
        embedding_model: str = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2",
        collection_name: str = "vietnamese_docs",
```

```
        persist_dir: str = "/content/chroma_data",
    ):
        self.persist_dir = persist_dir
        self.collection_name = collection_name

        self.embedding = HuggingFaceEmbeddings(model_name=embedding_model)
        self.db = self._build_db(documents)

    # Xây dựng hoặc tải Vector Database
    def _build_db(self, documents):
        # Trường hợp tải database đã có từ persist_dir
        if documents is None or len(documents) == 0:
            db = Chroma(
                collection_name=self.collection_name,
                embedding_function=self.embedding,
                persist_directory=self.persist_dir,
            )

        # Trường hợp tạo mới database từ documents
        else:
            db = Chroma.from_documents(
                documents=documents,
                embedding=self.embedding,
                collection_name=self.collection_name,
                persist_directory=self.persist_dir,
            )
        return db

    # Tạo retriever để tìm kiếm tương đồng
    def get_retriever(self, search_kwargs: dict = None):
        if search_kwargs is None:
            search_kwargs = {"k": 4} # Mặc định lấy 4 document gần nhất

        return self.db.as_retriever(
            search_type="similarity",
            search_kwargs=search_kwargs,
        )
```

IV.5. Khởi tạo LLM và xây dựng RAG Chain

Ta khởi tạo mô hình ngôn ngữ để đảm nhiệm phần sinh câu trả lời, đồng thời định nghĩa một RAG chain kết hợp retriever, prompt và LLM thành một quy trình thống nhất để trả lời câu hỏi dựa trên ngữ cảnh đã truy xuất.

Khởi tạo LLM

Hàm `get_hf_llm` khởi tạo một mô hình ngôn ngữ từ HuggingFace và đóng gói thành LangChain pipeline. Ở đây, ta sử dụng Qwen2.5-3B-Instruct, một mô hình instruction-tuned với khả năng xử lý hiệu quả trong các tác vụ hỏi đáp và sinh văn bản.

```python
import torch
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
from langchain_huggingface import HuggingFacePipeline

def get_hf_llm(
    model_name: str = "Qwen/Qwen2.5-3B-Instruct",
    temperature: float = 0.2, max_new_tokens: int = 450,
    **kwargs
):
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype=torch.float16,
        device_map="auto",
        low_cpu_mem_usage=True
    )

    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # Tạo text generation pipeline
    model_pipeline = pipeline(
        "text-generation",
        model=model,
        tokenizer=tokenizer,
        temperature=temperature,
        max_new_tokens=max_new_tokens,
        pad_token_id=tokenizer.eos_token_id,
        do_sample=True,
        top_p=0.75
    )

    # Đóng gói pipeline thành LangChain LLM
    llm = HuggingFacePipeline(pipeline=model_pipeline, model_kwargs=kwargs)
    return llm
```

Định nghĩa RAG Chain và parser

Class `FocusedAnswerParser` giúp làm sạch và rút gọn câu trả lời của LLM, loại bỏ các phần thừa hoặc lặp lại. Trong khi đó, class OfflineRAG kết nối các thành phần retriever, prompt template và LLM thành một chain hoàn chỉnh.

```python
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

class FocusedAnswerParser(StrOutputParser):
    def parse(self, text: str) -> str:
        text = text.strip()
        if "[TRẢ LỜI]:" in text:
            answer = text.split("[TRẢ LỜI]:")[-1].strip()
        else:
```

```
            answer = text

        answer = re.sub(r'^\s*[\\-\*]\s*', '', answer, flags=re.MULTILINE)
        answer = re.sub(r'\n+', ' ', answer)
        lines = [line.strip() for line in answer.split('. ') if line.strip() and len(
                                            line.strip()) > 5]
        return '. '.join(lines[:5]) + ('.' if lines else '')

class OfflineRAG:
    def __init__(self, llm):
        self.llm = llm
        self.prompt = PromptTemplate.from_template("""
Bạn là trợ lý AI phân tích tài liệu tiếng Việt.

[TÀI LIỆU]:
{context}

[CÂU HỎI]:
{question}

Hãy trả lời dựa trên tài liệu. Nếu tài liệu không có thông tin, nói rõ "Không có thông
                                            tin".
Trả lời đầy đủ thông tin (3-5 câu chi tiết), không thêm bất kỳ chi tiết nào ngoài tài
                                            liệu.
[TRẢ LỜI]:""")

        self.answer_parser = FocusedAnswerParser()

    # Xây dựng RAG chain từ retriever
    def get_chain(self, retriever):
        def format_docs(docs):
            formatted = []
            seen = set()
            for doc in docs:
                content = doc.page_content.strip()
                if content and len(content) > 40 and content not in seen:
                    formatted.append(content)
                    seen.add(content)
            return "\n\n".join(formatted)
        rag_chain = (
            {"context": retriever | format_docs, "question": RunnablePassthrough()}
            | self.prompt
            | self.llm
            | self.answer_parser
        )
        return rag_chain
```

Kết nối toàn bộ pipeline

Sau khi đã có đủ các thành phần, ta kết nối chúng lại thành một hàm xử lý duy nhất. Sau đó sử dụng hàm `answer_question` thực hiện toàn bộ quy trình: nhận câu hỏi, truy xuất dữ liệu,

xây dựng prompt và sinh câu trả lời cuối cùng.

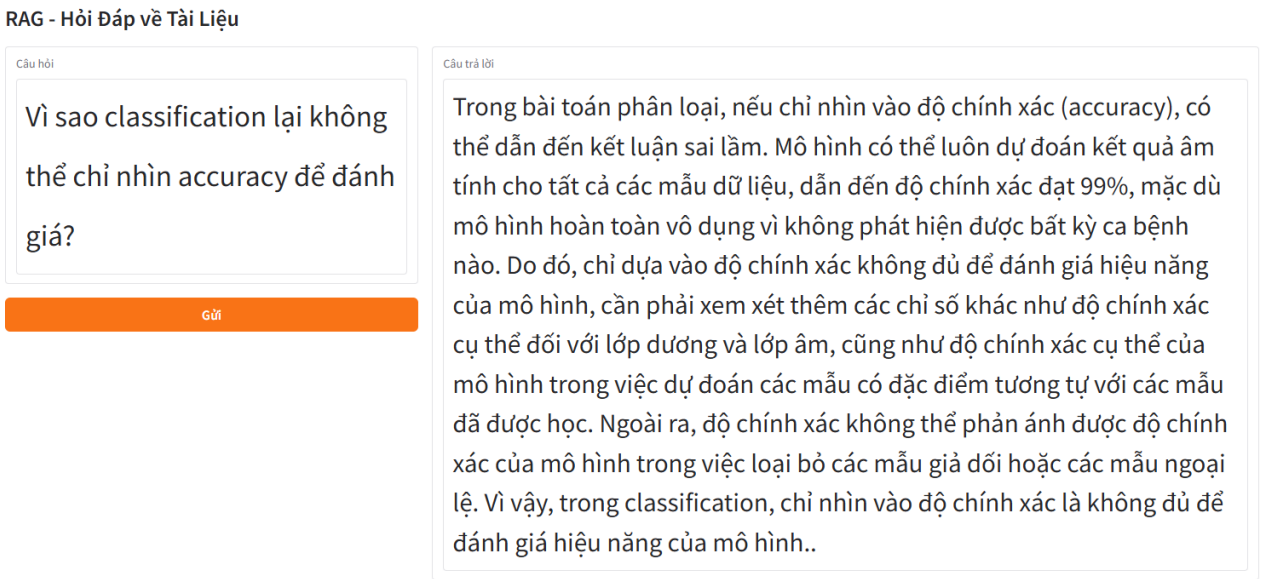
```
os.chdir("/content/rag_langchain")

# Khởi tạo LLM
llm = get_hf_llm()

data_dir = "/content/rag_langchain/data_source/generative_ai"

# Load và xử lý documents
loader = SimpleLoader()
text_splitter = TextSplitter(chunk_size=400, chunk_overlap=120)

raw_docs = loader.load_dir(data_dir)
split_docs = text_splitter.split(raw_docs)

# Xây dựng Vector Database và Retriever
vdb = VectorDB(documents=split_docs)
retriever = vdb.get_retriever(search_kwargs={"k": 4})

# Tạo RAG Chain
rag = OfflineRAG(llm)
rag_chain = rag.get_chain(retriever)

# Hàm xử lý câu hỏi
def answer_question(question: str) -> str:
    try:
        return rag_chain.invoke(question)
    except Exception as e:
        return f"Error: {str(e)}"
```

IV.6. Xây dựng giao diện ứng dụng

Cuối cùng, chúng ta xây dựng một giao diện đơn giản sử dụng Gradio để người dùng có thể tương tác trực tiếp với hệ thống RAG. Giao diện này không chỉ tạo điều kiện thuận lợi cho việc trải nghiệm hệ thống mà còn giúp minh họa trực quan toàn bộ quy trình hoạt động của pipeline RAG, từ việc tiếp nhận truy vấn đến khi trả về kết quả.

```
import gradio as gr

# Xây dựng giao diện với Gradio Blocks
with gr.Blocks(title="RAG Vietnamese QA") as demo:
    gr.Markdown("# RAG - Hỏi Đáp về Tài Liệu")

    # Thiết kế giao diện cho các thành phần hiển thị trên 1 dòng
    with gr.Row():
    # Cột bên trái: Input câu hỏi
        with gr.Column(scale=1):
            question_input = gr.Textbox(
```

```
                label="Câu hỏi",
                placeholder="Ví dụ: Vì sao classification lại không thể chỉ nhìn
                                        accuracy để đánh giá?",
                lines=3,
            )
            submit_btn = gr.Button("Gửi", variant="primary")

        # Cột bên phải: Output câu trả lời
        with gr.Column(scale=2):
            answer_output = gr.Textbox(
                label="Câu trả lời",
                lines=6,
                interactive=False
            )

    # Kết nối button với hàm xử lý
    submit_btn.click(
        fn=answer_question,
        inputs=question_input,
        outputs=answer_output,
    )

demo.launch(share=True)
```



Hình 11: Hình ảnh giao diện demo chương trình RAG

Hình 11 minh họa giao diện người dùng sau khi hệ thống được khởi tạo thành công. Khi chạy cell này, một đường link Gradio sẽ được tạo ra, cung cấp môi trường tương tác trực quan. Tại đây, người dùng có thể nhập câu hỏi liên quan đến các tài liệu đã được index, sau đó quan sát cách hệ thống RAG truy xuất ngữ cảnh và sinh ra câu trả lời tương ứng.

V. Câu hỏi trắc nghiệm

1. Trong cơ chế Hybrid Search, tại sao chúng ta không thể cộng trực tiếp điểm số từ Vector Search và Keyword Search mà phải cần đến thuật toán Reciprocal Rank Fusion (RRF)?

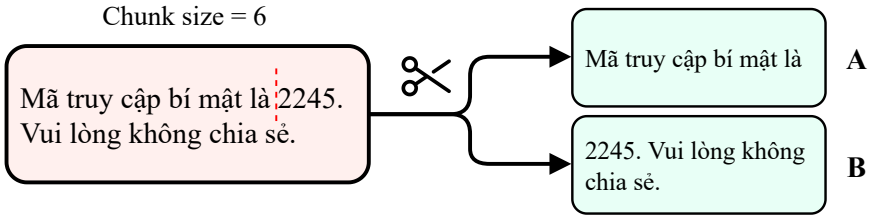
   (a) Vì Vector Search trả về kết quả quá chậm so với Keyword Search nên cần RRF để đồng bộ hóa thời gian.

   (b) Vì thang đo điểm số của hai phương pháp khác nhau, việc cộng trực tiếp sẽ khiến BM25 lấn át hoàn toàn Vector.

   (c) Vì Vector Search chỉ tìm được từ đồng nghĩa, trong khi RRF giúp loại bỏ các từ đồng nghĩa sai.

   (d) Vì RRF là thuật toán duy nhất có thể chạy trên GPU để tăng tốc độ xử lý.

2. Quan sát hình minh họa dưới đây:



   Nếu người dùng hỏi: *"Mã truy cập bí mật là gì?"*, kết quả nào sau đây phản ánh chính xác nhất hoạt động của hệ thống RAG?

   (a) Hệ thống tìm thấy Chunk B vì nó chứa con số cụ thể, và LLM trả lời chính xác mã số.

   (b) Hệ thống tìm thấy Chunk A, nhưng LLM sẽ không trả lời được mã số vì thông tin này nằm ở Chunk B, vốn không được tìm thấy.

   (c) Hệ thống sẽ tự động tìm thấy cả Chunk A và Chunk B nhờ cơ chế suy luận bắc cầu của Vector Database.

   (d) Hệ thống không tìm thấy cả hai chunk vì câu văn bị cắt đôi làm hỏng cấu trúc ngữ pháp.

3. Tại sao mô hình Cross-Encoder lại có độ chính xác cao hơn Bi-Encoder, mặc dù Cross-Encoder chậm hơn rất nhiều?

   (a) Vì Cross-Encoder quan sát được sự tương tác trực tiếp giữa từng token của câu hỏi và từng token của tài liệu cùng lúc.

   (b) Vì Cross-Encoder nén câu hỏi và tài liệu thành các vector có số chiều lớn hơn.

   (c) Vì Cross-Encoder sử dụng cơ sở dữ liệu đồ thị Graph Database thay vì Vector Database.

   (d) Vì Cross-Encoder thực hiện việc tìm kiếm từ khóa chính xác thay vì tìm kiếm ngữ nghĩa.

4. Để khắc phục hiện tượng "Lost in the Middle", chiến lược sắp xếp ngữ cảnh nào sau đây là tối ưu nhất trước khi gửi cho LLM?

(a) Sắp xếp theo thứ tự độ tương đồng giảm dần.

(b) Sắp xếp ngẫu nhiên để tránh việc mô hình bị thiên kiến vào vị trí.

(c) Sắp xếp theo hình chữ U: Đưa các tài liệu quan trọng nhất vào đầu và cuối prompt, giấu các tài liệu ít quan trọng hơn vào giữa.

(d) Sắp xếp theo trình tự thời gian: Các tài liệu mới nhất được đưa lên đầu, cũ nhất xuống cuối.

5. Trong LangChain, "đơn vị cơ bản" để biểu diễn thông tin là gì?

(a) Chunk.

(b) Vector.

(c) PromptTemplate.

(d) Document.

6. Trong Ví dụ 1 ở phần **Loading Documents**, PyPDFLoader tạo ra Document theo đơn vị nào?

(a) Mỗi file PDF là một Document.

(b) Mỗi trang trong file PDF là một Document.

(c) Mỗi chunk là một Document.

(d) Mỗi câu là một Document.

7. Đối tượng nào biến các chunk thành vector embedding để lưu/tra cứu trong VectorStore?

```
docs = PyPDFLoader("sample.pdf").load()

splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = splitter.split_documents(docs)

emb = OpenAIEmbeddings(model="text-embedding-3-large")
vs = Chroma.from_documents(chunks, embedding=emb)

retriever = vs.as_retriever(search_type="similarity", search_kwargs={"k": 4})
```

(a) PyPDFLoader.

(b) RecursiveCharacterTextSplitter.

(c) OpenAIEmbeddings.

(d) Chroma (Vector Store).

8. Quan sát đoạn code cấu hình Text Splitter dưới đây:

```
class TextSplitter:
    def init(self, chunk_size=400, chunk_overlap=120):
        self.splitter = RecursiveCharacterTextSplitter(
```

```
            separators=["\n\n", "\n", " ", ""],
            chunk_size=chunk_size,
            chunk_overlap=chunk_overlap,
            length_function=len,
        )
```

Trong bối cảnh xây dựng RAG, vai trò cụ thể của tham số `chunk_overlap=120` trong đoạn code trên là gì?

(a) Chia văn bản thành các đoạn nhỏ có kích thước cố định đúng bằng 120 ký tự bất kể cấu trúc câu.

(b) Tự động loại bỏ 120 ký tự đặc biệt hoặc khoảng trắng thừa ở đầu mỗi đoạn văn.

(c) Duy trì sự liên tục về ngữ cảnh giữa các đoạn liền kề bằng cách lặp lại 120 ký tự cuối của chunk trước sang chunk sau.

(d) Giới hạn độ dài tối đa của mỗi chunk là 120 ký tự để giảm tải bộ nhớ khi embedding.

9. Khi khởi tạo pipeline sinh văn bản cho hệ thống RAG, đoạn code cấu hình thường được thiết lập như sau:

```
model_pipeline = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    temperature=0.2,        # Lưu ý tham số này
    max_new_tokens=450,
    do_sample=True,
    top_p=0.75
)
```

Trong ngữ cảnh của bài toán RAG, tại sao chiến lược đặt `temperature` ở mức thấp lại là tiêu chuẩn được khuyến nghị thay vì đặt ở mức cao?

(a) Để ngăn chặn mô hình sinh ra các vòng lặp vô tận khi gặp các đoạn văn bản khó.

(b) Để tăng tính đa dạng cho câu trả lời, giúp mô hình có thể diễn đạt cùng một ý theo nhiều văn phong khác nhau trong mỗi lần chạy.

(c) Để giảm thiểu sự sáng tạo ngẫu nhiên, buộc mô hình tập trung vào các từ có xác suất cao nhất nhằm đảm bảo câu trả lời bám sát vào tài liệu được cung cấp.

(d) Để tăng tốc độ xử lý của mô hình, vì giá trị temperature cao đòi hỏi nhiều tài nguyên tính toán hơn để sinh ra các từ vựng phong phú.

10. Trước khi đưa các tài liệu tìm được vào Prompt, hàm helper `format_docs` thực hiện một bước lọc dữ liệu như sau:

```
def format_docs(docs):
    formatted = []
```

```
    seen = set()
    for doc in docs:
        content = doc.page_content.strip()
        if content and len(content) > 40 and content not in seen:
            formatted.append(content)
            seen.add(content)
    return "\n\n".join(formatted)
```

Mục đích thực tế của điều kiện `len(content) > 40` và `content not in seen` trong đoạn code trên là gì?

(a) Đảm bảo chỉ có tối đa 40 tài liệu quan trọng nhất được gửi vào mô hình để xử lý.

(b) Loại bỏ các đoạn văn bản quá ngắn, thiếu ngữ nghĩa và ngăn chặn việc lặp lại thông tin để tối ưu hóa chất lượng Prompt.

(c) Kiểm tra xem tài liệu có chứa ít nhất 40 từ khóa quan trọng liên quan đến câu hỏi hay không.

(d) Giới hạn tổng độ dài của prompt không vượt quá 40 token để tiết kiệm chi phí API và tránh lỗi Context Window.

Phụ lục

1. **Solution:** File code cài đặt thực nghiệm có thể được tải tại đây.
2. **Q&A:** Bạn có thể đặt thêm câu hỏi về nội dung bài đọc trong group Facebook hỏi đáp tại đây. Tất cả câu hỏi sẽ được trả lời trong vòng tối đa 4 tiếng.

Hình 12: Hình ảnh group facebook AIO Q&A.