



HƯỚNG DẪN SỬ DỤNG
LLAMAINDEX QUA DỰ ÁN CHĂM
SÓC SỨC KHỎE TINH THẦN

Dinh-Tiem Nguyen và Quang-Vinh Dinh

Mục lục

1 Giới thiệu	2
2 Phương pháp	3
2.1 Mô tả bài toán	3
2.2 Dữ liệu	5
2.3 Quy trình xử lý dữ liệu	5
2.4 Quy trình tạo Agent	6
2.5 Xây dựng ứng dụng	7
2.6 Quy trình đánh giá hệ thống	8
3 Các thành phần của Llammaindex	9
3.1 Yêu cầu trước khi bắt đầu	9
3.2 Documents	9
3.3 Nodes	11
3.4 Indexes	14
3.5 Truy vấn dữ liệu	15
4 Kỹ thuật Ingesting Data	18
4.1 Loading Data	18
4.2 Các kỹ thuật tạo node	20
4.3 Khai thác sức mạnh của metadata - Metadata Extraction	23
4.4 Ingestion Pipeline	28
4.5 Thực hành 1: Xây dựng Ingestion Pipeline - Mental Health	29
5 Kỹ thuật Indexing Data	34
5.1 Tạo Vector Store Index	34
5.2 Lưu trữ Index	35
5.3 Thực hành 2: Xây dựng Indexing - Mental Health	36
6 Kỹ thuật truy xuất dữ liệu	38
7 Kỹ thuật xây dựng Chatbot, Agent	41
7.1 Chat engine	41
7.2 Hoạt động của bộ nhớ trò chuyện	44
7.3 Các cơ chế trò chuyện	45
7.4 Agent	47
7.5 Thực hành 3: Xây dựng Agent - Mental Health	52
8 Kỹ thuật prompt	55
8.1 Prompts cho Metadata Extractors	55
8.2 Prompts cho cuộc trò chuyện	56
9 Evaluation	57
9.1 Thực hành 4: Đánh giá hệ thống Mental Health	60
10 Kết luận	62

1 Giới thiệu

Tại Việt Nam, sức khỏe tinh thần từ lâu đã bị coi là một vấn đề thứ yếu, chỉ dành cho những người có điều kiện kinh tế khá giả, hoặc thậm chí bị gán nhãn là vấn đề chỉ của những kẻ "điên dại". Sự kỳ thị xã hội này đã dẫn đến việc xem nhẹ nhu cầu chăm sóc sức khỏe tinh thần, gây ra những hậu quả nghiêm trọng về cả thể chất lẫn tinh thần cho hàng triệu người.

Theo thông tin từ báo [suckhoedoisong.vn](#) năm 2023, Việt Nam có khoảng 14 triệu người bị rối loạn tâm thần. Con số này thể hiện một tỷ lệ đáng báo động, đặc biệt là khi so sánh với số lượng chuyên gia tâm lý lâm sàng và tâm lý trị liệu chỉ dừng lại ở con số 143 người. Tỷ lệ này tương đương với khoảng 1 chuyên gia cho mỗi 100.000 người dân bị rối loạn tâm thần, cho thấy một sự thiếu hụt trầm trọng trong việc cung cấp dịch vụ chăm sóc sức khỏe tâm thần tại Việt Nam.

Trong khi dịch vụ chăm sóc sức khỏe tâm thần chủ yếu là điều trị bằng thuốc, thì những phương pháp hỗ trợ tâm lý chuyên sâu lại chưa được phổ biến rộng rãi. Hơn nữa, dịch vụ tâm lý lâm sàng chưa được bảo hiểm y tế chi trả, khiến nhiều người bệnh không tiếp cận được dịch vụ cần thiết. Điều này càng làm trầm trọng thêm khoảng cách giữa nhu cầu và khả năng cung cấp dịch vụ chăm sóc sức khỏe tâm thần.

Ngoài ra, nước ta đang phải đối mặt với nhiều thách thức mới, bao gồm già hóa dân số, tỷ lệ người khuyết tật kinh cao, và những ảnh hưởng sâu rộng từ đại dịch COVID-19. Đại dịch đã làm gia tăng tỷ lệ lo âu và trầm cảm lên hơn 25% chỉ trong năm đầu tiên, điều này cho thấy sức khỏe tâm thần đang trở thành một vấn đề ngày càng nghiêm trọng.

Trước tình hình này, bài viết hướng đến mục tiêu ứng dụng công nghệ tiên tiến với chi phí thấp để xây dựng hệ thống chăm sóc sức khỏe tinh thần thông minh. Hệ thống này đóng vai trò như một chuyên gia tâm lý, có thể trò chuyện với con người một cách tự nhiên và có khả năng đánh giá, theo dõi sức khỏe tinh thần của người dùng qua kiến thức chuyên môn từ DSM5 -Sổ tay chẩn đoán và thống kê các rối loạn tâm thần.

Việc xây dựng một hệ thống chăm sóc sức khỏe tinh thần thông minh không chỉ là một bước đi cần thiết mà còn là một giải pháp kịp thời để giảm bớt gánh nặng cho hệ thống y tế. Nó có thể hỗ trợ những người không có khả năng tiếp cận với các chuyên gia tâm lý, cung cấp hỗ trợ tâm lý kịp thời và phổ biến thông tin về sức khỏe tâm thần, giúp giảm bớt sự kỳ thị và nâng cao nhận thức của cộng đồng.

Và tất nhiên rồi, bài viết này dành cho các bạn AIO và cộng đồng AI VIET NAM, vì vậy hướng dẫn sử dụng công nghệ và xây dựng hệ thống là một trong những mục tiêu hàng đầu. Nội dung của bài viết này sẽ đan xen giữa lý thuyết và thực hành, đủ cho bạn có thể phát triển các hệ thống của riêng mình để giải quyết các vấn đề khác, mà không bị giới hạn trong dự án chăm sóc sức khỏe tinh thần. Nội dung chính của các phần như sau:

- Phần 1: Như đã trình bày, phần này nhằm giới thiệu bài toán, đặt vấn đề và nêu rõ những mục tiêu của bài viết.
- Phần 2: Trình bày phương pháp thực hiện, mô tả hệ thống và các thành phần cơ bản.
- Phần 3: Hướng dẫn sử dụng thư viện Llamaindex cơ bản, giới thiệu các thành phần của một hệ thống RAG cơ bản.
- Phần 4: Phần này sẽ đi sâu vào chi tiết các kỹ thuật load dữ liệu, tạo nodes và khai thác tiềm năng của metadata. Sau đó hướng dẫn sử dụng Ingest pipeline để tạo quy trình hoàn chỉnh cho quá trình xử lý dữ liệu. Cuối phần này là phần thực hành, chúng ta sẽ tiến hành tạo Ingest Pipeline cho dự án chăm sóc sức khỏe tinh thần.
- Phần 5 Kỹ thuật Indexing Data: Phần này sẽ đi sâu vào chi tiết các kỹ thuật tạo Index, bao gồm việc tạo Vector Store Index và Vector Database index. Cuối cùng là phần thực hành xây dựng Index cho dự án chăm sóc sức khỏe tinh thần.

- Phần 6 Kỹ thuật truy xuất dữ liệu: Phần này trình bày cơ chế của hoạt động và hướng dẫn tinh chỉnh các thành phần trong hệ thống truy xuất dữ liệu.
- Phần 7 Kỹ thuật xây dựng Chatbot, Agent: Trong phần này đi sâu vào các loại công cụ và cơ chế quản lý cuộc hội thoại. Hướng dẫn tạo Agent thông minh và khai thác tiềm năng của chúng. Cuối phần này, chúng ta sẽ thực hành tạo Agent cho hệ thống chăm sóc sức khỏe tinh thần.
- Phần 8 Kỹ thuật prompt: Phần này giới thiệu và giải thích cấu trúc của các loại prompt sử dụng trong llamaindex, sau phần này bạn có thể tạo được prompt cho riêng mình.
- Phần 9 Đánh giá hệ thống - Evaluate RAG: Trong phần này sẽ trình bày kỹ thuật để đánh giá hệ thống RAG, các độ đo cơ bản và tập trung vào đánh giá thành phần query engine. Cuối phần này chúng ta sẽ thực hành đánh giá hệ thống chăm sóc sức khỏe tinh thần.
- Phần 10 Kết Luận: Tóm lại một số ý chính, nhận xét, hướng phát triển đề xuất.

Mã nguồn chi tiết tại: <https://github.com/NguyenDinhTiem/AIO-MENTAL-HEALTH.git>

2 Phương pháp

Với sự phát triển mạnh mẽ của công nghệ trí tuệ nhân tạo, các mô hình ngôn ngữ lớn (Large Language Models - LLM) hiện nay có khả năng hiểu và tạo ra văn bản với mức độ tự nhiên như con người. Tuy nhiên, một vấn đề lớn khi sử dụng các LLM là hiện tượng ảo giác (hallucination), khi mô hình đưa ra thông tin không chính xác hoặc không có trong dữ liệu huấn luyện. Hơn nữa, các LLM thường không được cập nhật dữ liệu thường xuyên, dẫn đến việc phản hồi dựa trên thông tin lỗi thời.

Để khắc phục các vấn đề trên, giải pháp Retrieval-Augmented Generation (RAG) đã được phát triển. RAG kết hợp sức mạnh của các LLM với hệ thống truy xuất thông tin từ một cơ sở dữ liệu hoặc tài liệu bên ngoài. Khi nhận được câu hỏi, hệ thống RAG sẽ tìm kiếm và lấy thông tin liên quan từ các nguồn dữ liệu đáng tin cậy, sau đó sử dụng LLM để tạo ra câu trả lời dựa trên thông tin được truy xuất. Phương pháp này giúp giảm thiểu ảo giác và tăng độ chính xác của phản hồi.

Có nhiều thư viện hỗ trợ xây dựng hệ thống RAG, trong bài viết này, chúng ta sẽ sử dụng LlamaIndex - là một thư viện giúp xây dựng các LLM có khả năng thích ứng với các trường hợp cụ thể. Thay vì chỉ dựa vào kiến thức chung chung đã được mô hình đào tạo trước, LlamaIndex cho phép chúng ta kết hợp LLM với dữ liệu bên ngoài theo nhu cầu cụ thể của từng ứng dụng. Thư viện này hỗ trợ tích hợp dữ liệu từ nhiều nguồn khác nhau, giúp mô hình đưa ra các phản hồi chính xác và phù hợp hơn với ngữ cảnh cụ thể.

2.1 Mô tả bài toán

Chúng ta sẽ thực hiện phát biểu lại bài toán, xác định các chức năng và luồng hoạt động cơ bản của hệ thống cần xây dựng.

Bài toán: Phát triển một hệ thống chăm sóc sức khỏe tinh thần có khả năng hỗ trợ người dùng trong việc trò chuyện, tư vấn tâm lý, phân tích và chẩn đoán tình trạng sức khỏe tâm thần theo DSM5, cũng như theo dõi tiến trình sức khỏe của họ theo thời gian.

Các yêu cầu chức năng:

- Trò chuyện và tư vấn tâm lý: Hệ thống phải có khả năng thực hiện các cuộc trò chuyện tự nhiên, thân thiện và đồng cảm với người dùng. Nó có thể cung cấp các tư vấn tâm lý cơ bản dựa trên thông tin được thu thập từ người dùng, giúp họ giải tỏa cảm xúc và định hướng giải quyết các vấn đề tâm lý.

- Phân tích và chẩn đoán: Hệ thống cần có khả năng phân tích các tương tác với người dùng để đánh giá và chẩn đoán sơ bộ về tình trạng sức khỏe tâm thần của họ, bao gồm các bệnh lý tâm thần phổ biến (như lo âu, trầm cảm) và phân loại mức độ tình trạng (tốt, khá, trung bình, kém), dựa trên dữ liệu DSM5.
- Theo dõi tiến trình sức khỏe: Hệ thống cần theo dõi tiến trình sức khỏe tâm thần của người dùng theo thời gian, lưu trữ lịch sử tương tác và các chỉ số liên quan. Dựa trên dữ liệu này, Hệ thống cung cấp các khuyến nghị điều chỉnh hành vi hoặc nhắc nhở người dùng duy trì thói quen tốt cho sức khỏe tâm thần.

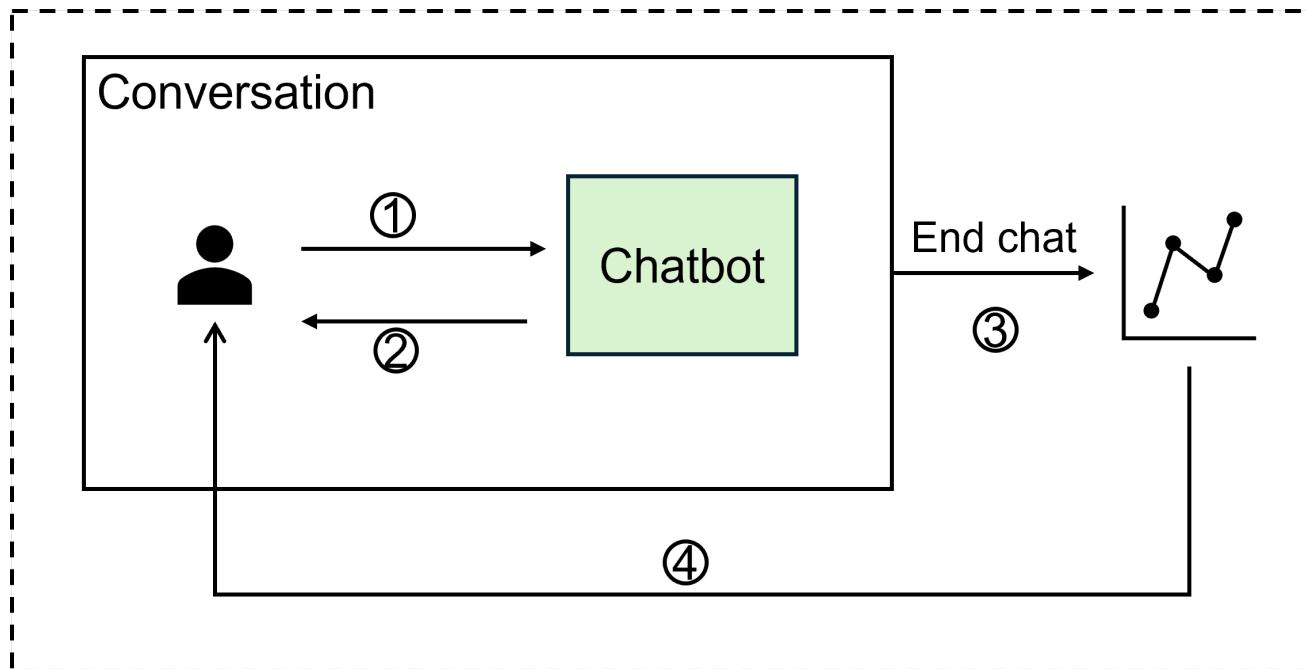
Các yêu cầu phi chức năng:

- Tính bảo mật và quyền riêng tư: Dữ liệu sức khỏe tâm thần của người dùng phải được bảo vệ nghiêm ngặt, tuân thủ các quy định về bảo mật thông tin cá nhân.
- Tính khả dụng: Hệ thống cần hoạt động liên tục, sẵn sàng hỗ trợ người dùng 24/7.
- Tính dễ sử dụng: Giao diện người dùng phải đơn giản, thân thiện và dễ dàng truy cập, phù hợp với mọi đối tượng người dùng.

Thời gian và chi phí:

- Thời gian: ...
- Chi phí: ...

Dựa vào đặc tả bài toán, chúng ta mô phỏng hệ thống ban đầu như sơ đồ dưới đây:

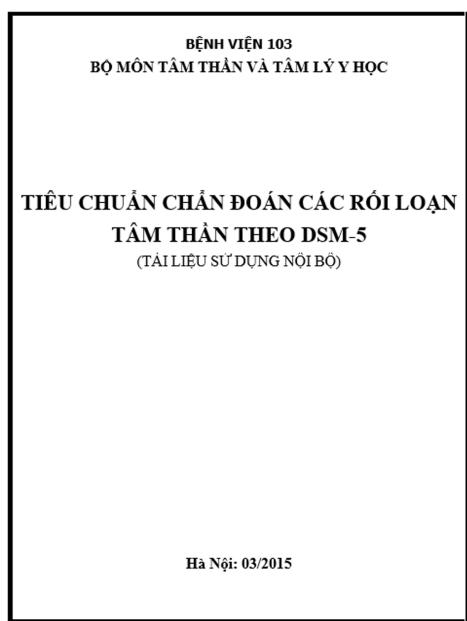


Hình 1: Hệ thống thực hiện trò chuyện với người dùng (1)(2), phân tích và chẩn đoán (3), theo dõi tiến trình sức khỏe (4)

2.2 Dữ liệu

Trong lĩnh vực chẩn đoán và phân loại các rối loạn tâm thần, DSM-5 (Diagnostic and Statistical Manual of Mental Disorders, 5th Edition) là tài liệu chuẩn mực được sử dụng rộng rãi trên toàn thế giới. Với hơn 300 loại rối loạn tâm thần được liệt kê và mô tả chi tiết, DSM-5 cung cấp các tiêu chí chẩn đoán cụ thể. Tuy nhiên tài liệu này lên đến hơn 1000 trang, do muốn giảm chi phí và tối ưu hóa quá trình thử nghiệm, chúng ta sẽ chọn sử dụng một giáo trình rút gọn, là bản dịch tiếng Việt của DSM-5, mang tên "Tiêu chuẩn chẩn đoán các rối loạn tâm thần theo DSM-5." Tài liệu này gồm 106 trang và được biên soạn bởi Bệnh viện 103, bộ môn Tâm thần và Tâm lý học.

Giáo trình rút gọn này bao gồm 15 nhóm rối loạn tâm thần chính, chẳng hạn như rối loạn phát triển thần kinh, rối loạn trầm cảm, và nhiều loại rối loạn khác. Cấu trúc chung mỗi loại bệnh cụ thể được viết theo hai phần chính là: các tiêu chí chẩn đoán và hướng dẫn chẩn đoán. Các tiêu chí được trình bày một cách cụ thể, giúp xác định tình trạng sức khỏe tâm thần của bệnh nhân. Dựa trên những tiêu chí này, tài liệu cung cấp hướng dẫn cụ thể để thực hiện chẩn đoán và phân loại các rối loạn tâm thần một cách chính xác và hiệu quả.



Ví dụ:

Rối loạn Điều hòa Khí sắc

Tiêu chí:

- A. Các cơn bùng nổ cảm xúc trầm trọng tái diễn dai dẳng, thể hiện dưới dạng ngôn ngữ...

Chẩn đoán phân biệt:

- Rối loạn từng cơn...

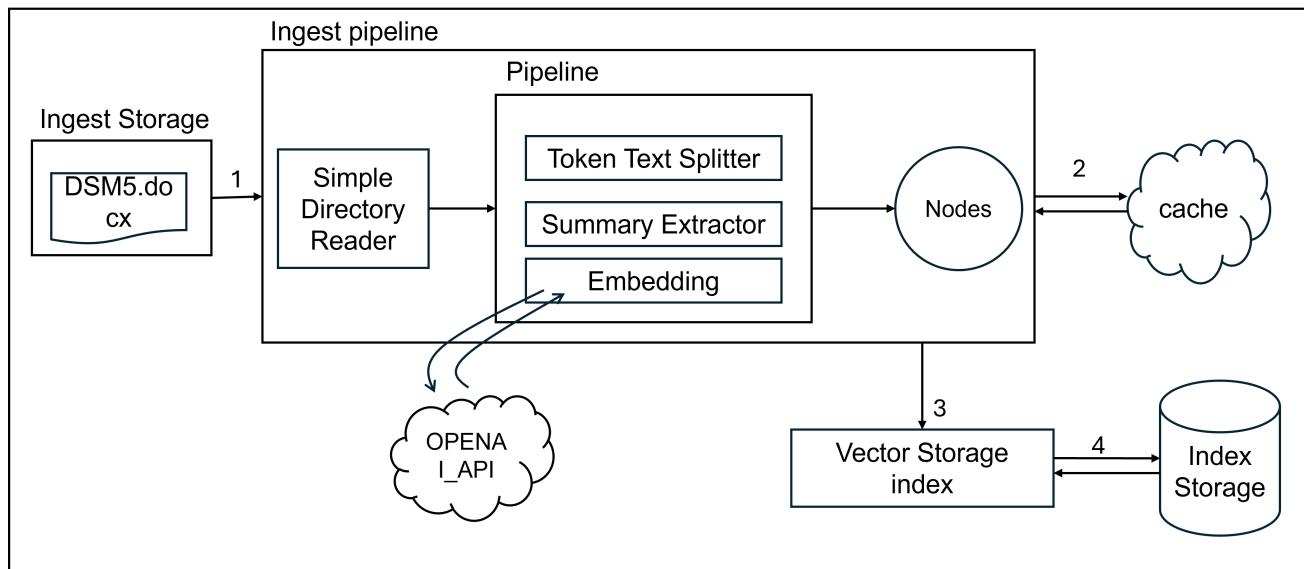
Hình 2: Giáo trình "Tiêu chuẩn chẩn đoán các rối loạn tâm thần theo DSM-5" và một mẫu dữ liệu về Rối loạn điều hòa khí sắc.

2.3 Quy trình xử lý dữ liệu

Quy trình xử lý dữ liệu của chúng ta bắt đầu từ kho dữ liệu thô, nơi các tài liệu được lưu trữ và chuẩn bị cho các bước xử lý tiếp theo. Đầu tiên, dữ liệu này được xử lý bởi Ingest Pipeline thông qua công cụ Simple Directory Reader, giúp tạo ra các đối tượng Documents từ dữ liệu thô.

Sau khi các Documents được tạo, chúng sẽ được đưa qua pipeline để chia nhỏ thành nhiều phần nhỏ hơn gọi là nodes, bằng cách sử dụng công cụ Token Text Splitter. Quá trình này giúp các tài liệu lớn được phân đoạn thành các đơn vị nhỏ hơn, dễ quản lý và xử lý hơn.

Tiếp theo, mỗi node sẽ được bổ sung thêm thông tin metadata bằng kỹ thuật Summary Extractor. Kỹ thuật này có nhiệm vụ tóm tắt nội dung của toàn bộ văn bản trong node hiện tại, và trong một số trường hợp, có thể bao gồm cả nội dung của node liền kề trước và sau. Điều này giúp cung cấp ngữ cảnh đầy đủ cho mỗi node, làm cho quá trình truy xuất thông tin sau này trở nên hiệu quả hơn.



Hình 3: Quy trình xử lí dữ liệu, dữ liệu thô được xử lí bởi Ingest Pipeline(1), quy trình xử lí được lưu lại tại bộ nhớ cache(2), sau đó quá trình tạo index và lưu trữ tại kho index(3)

Sau khi đã có metadata, chúng ta thực hiện embedding cho node hiện tại thông qua mô hình embedding được cung cấp bởi OPENAI API. Việc embedding này giúp chuyển đổi nội dung văn bản thành dạng vector để dễ dàng tìm kiếm và so sánh trong các bước tiếp theo.

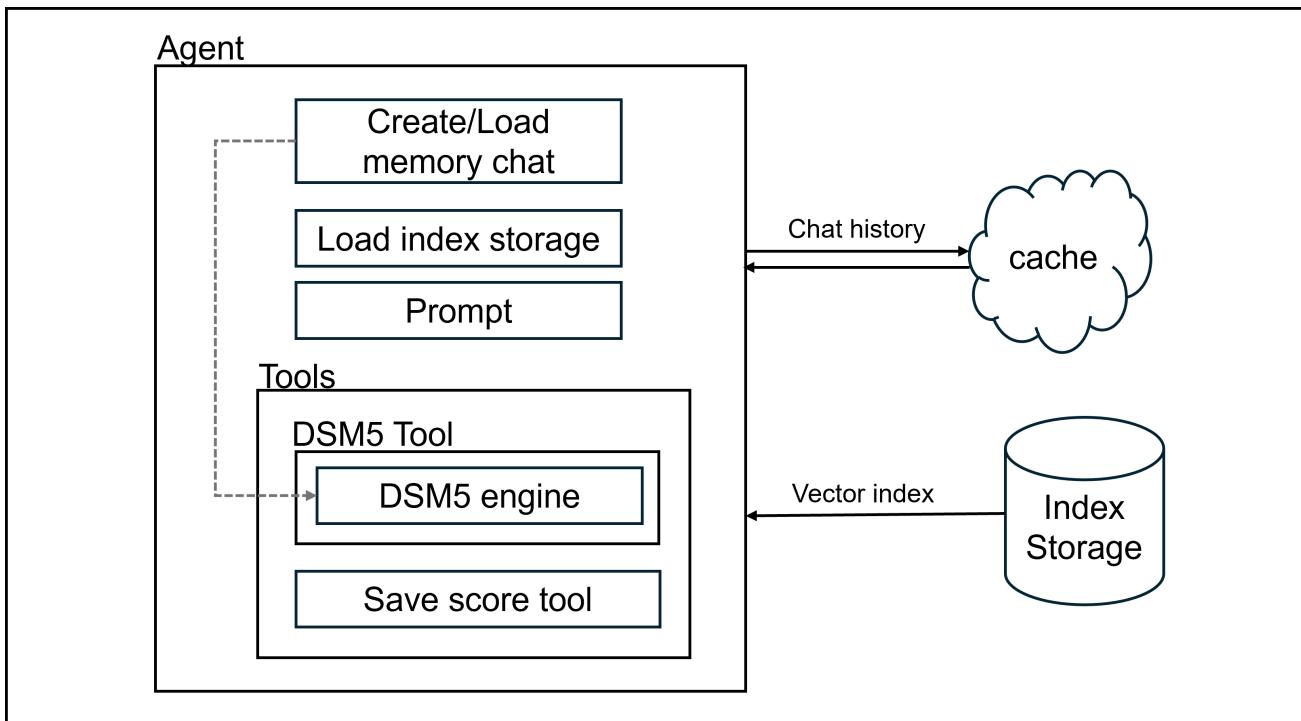
Toàn bộ quy trình tạo nodes này được lưu lại trong bộ nhớ cache để có thể tải lại nhanh chóng mà không cần phải thực hiện toàn bộ quy trình Ingest Pipeline từ đầu mỗi khi cần. Cuối cùng, từ các nodes đã được embedding, chúng ta tạo ra index và lưu trữ index này vào kho dữ liệu Index. Index này sẽ được sử dụng để truy xuất thông tin một cách hiệu quả và nhanh chóng trong các truy vấn sau này.

2.4 Quy trình tạo Agent

Trong hệ thống chăm sóc sức khỏe tinh thần, Agent đóng vai trò quan trọng như bộ não điều khiển toàn bộ hoạt động của hệ thống. Agent quản lý lịch sử của cuộc trò chuyện, khi cuộc trò chuyện bắt đầu, nó sẽ kiểm tra xem có thông tin lịch sử trò chuyện trước đó không, nếu có nó sẽ tiếp tục cuộc trò chuyện đó, nếu không nó sẽ bắt đầu tạo một bộ nhớ lưu trữ cuộc trò chuyện mới. Điều này sẽ giúp cuộc trò chuyện trở nên liền mạch và giúp theo dõi, đánh giá sức khỏe tinh thần của người dùng tốt hơn.

Chúng ta sẽ sử dụng prompt để thiết lập ngữ cảnh cho Agent, ngoài ra ta sẽ tạo hai công cụ để agent sử dụng đó là DSM5 tool - công cụ giúp truy xuất thông tin từ kho vector, và công cụ lưu trữ kết quả chẩn đoán. Về cơ bản thì Agent sẽ cần phải thực hiện các công việc sau:

- Đóng vai là chuyên gia tâm lí, nói chuyện với người dùng một cách tự nhiên
- Khi người dùng yêu cầu chẩn đoán, hoặc kết thúc trò chuyện hoặc khi Agent đã có đủ thông tin để chẩn đoán. Nó phải tiến hành chẩn đoán bằng cách tổng hợp lại cuộc trò chuyện của người dùng và tìm kiếm thông tin về các bệnh liên quan trong kho dữ liệu. Dựa vào thông tin tìm kiếm được và nội dung cuộc trò chuyện hiện tại, đưa ra tổng đoán và điểm số sức khỏe của người dùng theo các mức độ (tốt, khá, trung bình, kém)
- Kết quả chẩn đoán sẽ được lưu lại để phục vụ cho quá trình phân tích và theo dõi.



Hình 4: Agent quản lý lịch sử trò chuyện và sử dụng các công cụ truy vấn DSM5 và công cụ lưu trữ kết quả chẩn đoán.

2.5 Xây dựng ứng dụng

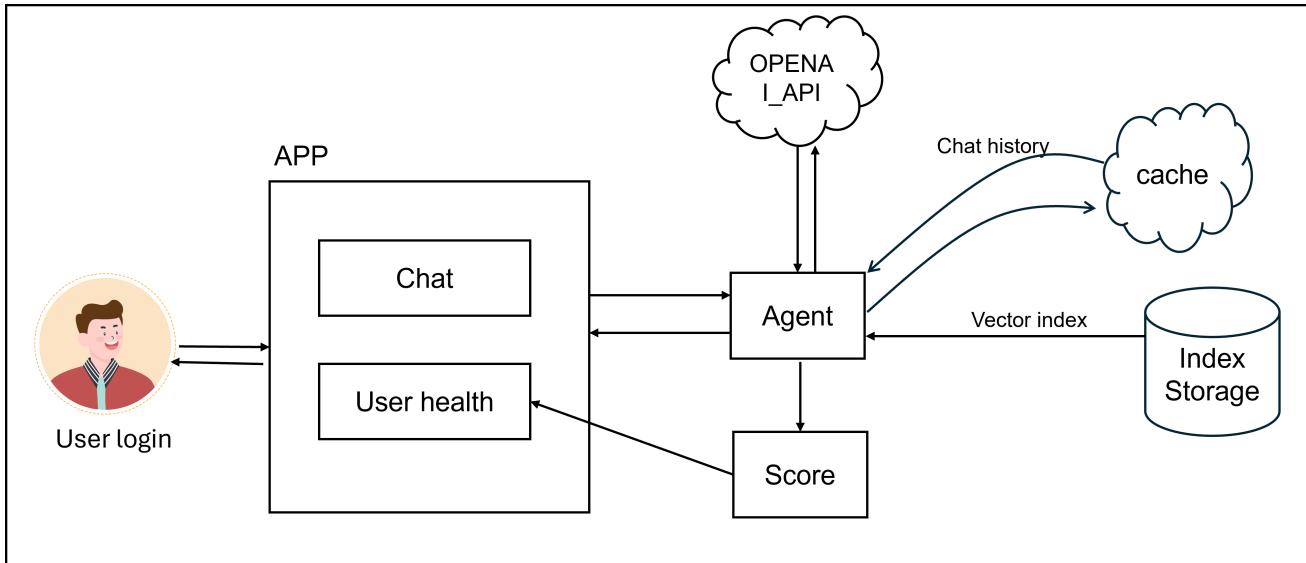
Trong dự án này, chúng ta sẽ sử dụng Streamlit—một nền tảng mã nguồn mở dành cho việc xây dựng và triển khai các ứng dụng web tương tác, đặc biệt phù hợp cho các dự án về khoa học dữ liệu và máy học—để thiết kế giao diện người dùng cho hệ thống chăm sóc sức khỏe tinh thần.

Streamlit được lựa chọn nhờ vào các ưu điểm nổi bật như khả năng xây dựng giao diện đơn giản, nhanh chóng và tính tương tác cao mà không cần nhiều kiến thức về phát triển web. Điều này cho phép chúng ta tập trung vào việc phát triển hệ thống RAG mà không phải lo lắng quá nhiều về các vấn đề giao diện phức tạp.

Về cơ bản, chúng ta sẽ cho phép người dùng đăng nhập, đăng ký tài khoản mới. Sau khi đăng nhập, ứng dụng cung cấp 2 tính năng mà người dùng có thể sử dụng là Chat để trò chuyện với chuyên gia AI, và User health để theo dõi kết quả, thống kê về tình trạng sức khỏe tinh thần của người dùng.

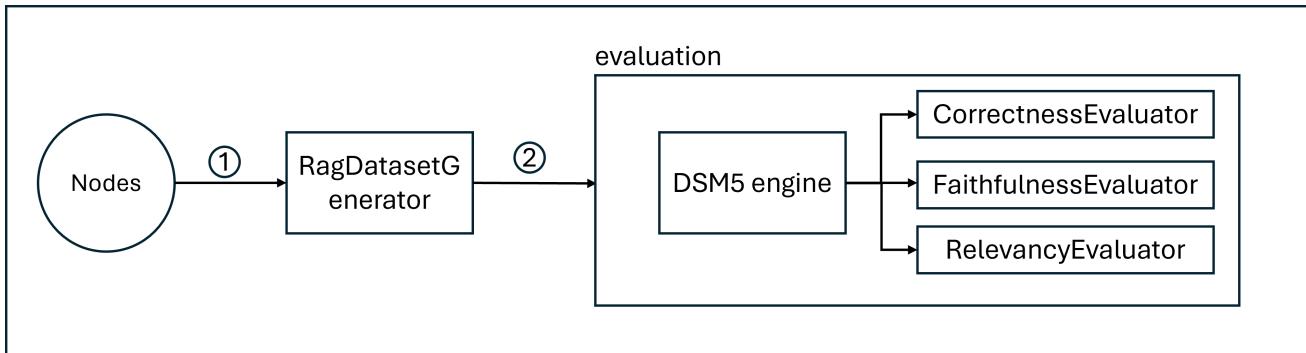
Với tính năng chat, người dùng bắt đầu cuộc trò chuyện bằng cách gửi nội dung tin nhắn, khi đó Agent sẽ nhận được tin nhắn của người dùng và tiến hành tạo câu trả lời. Sau đó Agent sẽ phản hồi lại tin nhắn của người dùng, cuộc trò chuyện cứ tiếp tục cho đến khi người dùng thoát khỏi cuộc trò chuyện. Lịch sử trò chuyện sẽ được lưu trữ lại và sẽ được tải lại sau khi người dùng quay lại ứng dụng.

Khi người dùng yêu cầu chẩn đoán, yêu cầu kết thúc cuộc trò chuyện, hoặc Agent đã có đủ thông tin để chẩn đoán. Agent sẽ tiến hành chẩn đoán và phản hồi lại tin nhắn kết quả chẩn đoán cho người dùng, đồng thời kết quả cũng được lưu lại. Khi người dùng sử dụng tính năng user health, ứng dụng sẽ hiển thị thống kê chuẩn đoán 7 ngày gần nhất và bảng kết quả chứa lịch sử chẩn đoán.



Hình 5: Luồng hoạt động của ứng dụng

2.6 Quy trình đánh giá hệ thống



Hình 6: Quy trình đánh giá hệ thống, tạo tập câu hỏi đánh giá (1), đánh giá trên 3 độ đo (2)

Đầu tiên, chúng ta tạo tập dữ liệu đánh giá bằng công cụ RagDatasetGenerator, công cụ này sẽ sử dụng LLM để sinh câu hỏi từ nội dung của từng node, tạo thành một tập dữ liệu các câu hỏi để đánh giá khả năng truy xuất của DSM5 engine. Sau đó sử dụng gói evaluation để thực hiện đánh giá trên ba độ đo Correctness, Faithfulness, Relevancy.

3 Các thành phần của Llammaindex



Hình 7: Các thành phần cơ bản Llammaindex

Trước khi bắt đầu xây dựng hệ thống phức tạp, chúng ta sẽ tìm hiểu kiến trúc của một hệ thống RAG cơ bản được xây dựng bởi Llama index sẽ gồm các thành phần nào. Sau phần này ta có thể tạo một ứng dụng RAG cơ bản. Code phần này có tại: [docs/tutorial/1.BasicLlamaindex.ipynb](#)

3.1 Yêu cầu trước khi bắt đầu

Để thực hành chạy lại các ví dụ, chúng ta cần thực hiện hướng dẫn sau.

1 `conda create -n aio_env python=3.11`

2. Cài đặt các thư viện

```

1 llama_index==0.11.4
2 PyYAML
3 plotly==5.24.0
4 streamlit==1.38.0
5 docx2txt==0.8
6 chromadb==0.5.5
7 llama-index-vector-stores-chroma==0.2.0
8 llama-index-extractors-entity==0.2.0
9 llama-index-readers-web==0.2.2
10 transformers==4.40.2

```

2. Tạo OpenAI API Key tại <https://platform.openai.com/api-keys>, sau khi tạo API chúng ta cần lưu lại để sử dụng sau.

3. Tạo Llammaindex Cloud API Key tại <https://cloud.llamaindex.ai/login>

4. Cài đặt git và tạo tài khoản github

3.2 Documents

Trong các ứng dụng RAG, chúng ta phải làm việc với nhiều loại dữ liệu khác nhau, từ các tệp văn bản như PDF, DOCX, đến dữ liệu từ cơ sở dữ liệu hay kết quả từ API. Những loại dữ liệu này thường có định dạng khác nhau, và điều này có thể gây khó khăn trong việc quản lý và xử lý.

Để giải quyết vấn đề này, LlamaIndex sử dụng một khái niệm gọi là Document. Document đóng vai trò như một "container" (bộ chứa), giúp chúng ta lưu trữ các loại dữ liệu khác nhau trong một cấu trúc chung. Điều này giúp cho việc quản lý và khai thác dữ liệu trở nên dễ dàng và hiệu quả hơn.

Một trong những điểm đặc biệt của Document là nó có thể chứa không chỉ dữ liệu chính (thường là văn bản), mà còn các thông tin bổ sung gọi là siêu dữ liệu (metadata). Siêu dữ liệu này có thể là các mô tả, tóm tắt, tiêu đề, hoặc bất kỳ thông tin nào khác mà chúng ta muốn gắn kèm với dữ liệu, giúp việc tìm kiếm và phân tích dữ liệu chính xác hơn.

Để tạo một Document trong LlamaIndex, chúng ta sử dụng cú pháp sau:

1 `Document(text, metadata, id_)`

Đây là những thành phần cơ bản nhất của Document, trong đó:

- text: Dữ liệu chính, thường ở dạng văn bản.

- metadata: Siêu dữ liệu, được lưu trữ dưới dạng từ điển (dictionary), có thể được tạo tự động bởi thư viện hoặc do chúng ta cung cấp.
- id_: ID của Document, thường được tự động tạo để định danh duy nhất mỗi Document.

Trong ví dụ này, chúng ta sẽ tạo một Document đơn giản trong LlamaIndex để lưu trữ một đoạn văn bản cùng với một số thông tin bổ sung.

```

1 from llama_index.core import Document
2
3 text = "AI VIET NAM"
4 doc = Document(
5     text=text,
6     metadata={
7         "fb": "fb/aivietnam.edu.vn"
8     },
9     id_="1"
10 )
11 print(doc)

```

```

=====
Output =====
Doc ID: 1
Text: AI VIET NAM
=====
```

Giải thích:

- Khai báo thư viện: Dòng đầu tiên from llama_index.core import Document nhập lớp Document từ thư viện LlamaIndex. Lớp này sẽ được sử dụng để tạo một Document mới.
- Tạo dữ liệu: Dòng text = "AI VIET NAM" khởi tạo một chuỗi văn bản đơn giản là "AI VIET NAM". Đây chính là nội dung chính mà chúng ta muốn lưu trữ trong Document. Thông thường dữ liệu thô thường nằm trong các định dạng file như pdf, csv... chúng ta cần sử dụng các công cụ phù hợp để đọc các file này, chúng ta sẽ được tìm hiểu các kỹ thuật này ở phần sau.
- Tạo Document: Tiếp theo, chúng ta tạo một Document mới bằng cách sử dụng Document(text=text, metadata={"fb": "fb/aivietnam.edu.vn"}, id_="1"). Ở đây, chúng ta truyền vào ba đối số:
 - text: Nội dung chính của Document, trong trường hợp này là "AI VIET NAM".
 - metadata: Một từ điển chứa các thông tin bổ sung, trong ví dụ này là một liên kết Facebook với khóa "fb" và giá trị "fb/aivietnam.edu.vn".
 - id_: Mã định danh duy nhất của Document, chúng ta đặt nó là 1.
- Cuối cùng, print(doc) in ra thông tin của Document mà chúng ta vừa tạo. Kết quả này sẽ hiển thị đoạn văn bản, siêu dữ liệu, và mã định danh mà chúng ta đã đặt cho Document.

Ngoài ra, chúng ta có thể xem toàn bộ cấu trúc của document bằng cách in mọi thông tin của nó ra như sau:

```

1 from llama_index.core import Document
2
3 text = "AI VIET NAM"
4 doc = Document(
5     text=text,
6     metadata={"fb": "fb/aivietnam.edu.vn"},
7     id_="1"
8 )
9 for i in doc:
10     print(i)

```

```
===== Output =====
    id_, '1')
('embedding', None)
('metadata', {'fb': 'fb/aivietnam.edu.vn'})
('excluded_embed_metadata_keys', [])
('excluded_llm_metadata_keys', [])
('relationships', {})
('text', 'AI VIET NAM')
('mimetype', 'text/plain')
('start_char_idx', None)
('end_char_idx', None)
('text_template', '{metadata_str}\n\n{content}')
('metadata_template', '{key}: {value}')
('metadata_separator', '\n')
=====
```

3.3 Nodes

Việc tạo Documents là bước đầu tiên, nhưng để dữ liệu thô này có thể được mô hình ngôn ngữ lớn (LLM) xử lý hiệu quả, chúng ta cần chia nhỏ nó ra. Đây là lúc Nodes phát huy vai trò. Nodes là các "mảnh" nhỏ hơn của một Document, như một đoạn văn bản hoặc hình ảnh, giúp chia tài liệu thành các phần dễ quản lý hơn.

Nodes giúp bạn tránh vượt quá giới hạn đầu vào của mô hình và giảm chi phí xử lý. Ví dụ, thay vì đưa cả cuốn ebook dài 100 trang vào một prompt, bạn có thể chia nó thành nhiều Nodes nhỏ hơn để mô hình có thể xử lý từng phần một cách chính xác và hiệu quả. Hơn nữa, các Nodes có thể được liên kết với nhau, giúp tổ chức và quản lý thông tin một cách logic.

Trong LlamaIndex, bạn có thể tạo Nodes từ dữ liệu văn bản bằng cách sử dụng lớp `TextNode`. Cú pháp sử dụng như sau:

```
1 from llama_index.core.schema import TextNode
2 TextNode(text)
```

Trong đó `text` là một phần văn bản trong Document

Dưới đây là một ví dụ minh họa việc tạo node đơn giản từ lớp `TextNode`:

```
1 from llama_index.core import Document
2 from llama_index.core.schema import
3     TextNode
4
5 text = "AI VIET NAM"
6 doc = Document(
7     text=text
8 )
9
9 node1 = TextNode(text=doc.text[:2])
10 node2 = TextNode(text=doc.text[3:])
11
12 print(node1)
13 print(node2)
```

===== Output =====

```
Node ID: 2079a5dc-1013-41c0-8ce2-600
cc5d72f84
Text: AI
Node ID: f975f87a-0da3-4c16-bd8a-7454
ce4d3101
Text: VIET NAM
```

Ví dụ trên minh họa cách chia nhỏ một Document thành các phần nhỏ hơn gọi là Nodes. Đầu tiên, chúng ta tạo một Document từ chuỗi văn bản "AI VIET NAM". Sau đó, chúng ta chia văn bản này thành hai phần: "AI" và "VIET NAM", mỗi phần được lưu trữ trong một `TextNode` riêng biệt. Các Nodes này giúp mô hình xử lý dữ liệu từng phần một cách dễ dàng và hiệu quả hơn.

Trong ví dụ trước, chúng ta đã tạo các Nodes một cách thủ công bằng cách chia nhỏ văn bản trong Document. Tuy nhiên, quá trình này có thể được thực hiện tự động bằng cách sử dụng TokenTextSplitter. Cú pháp sử dụng:

```

1 from llama_index.core.node_parser import TokenTextSplitter
2
3 TokenTextSplitter(
4     chunk_size,
5     chunk_overlap,
6     separator,
7 )

```

Giải thích tham số:

- chunk_size: Đây là kích thước của mỗi đoạn văn bản (chunk) sau khi chia nhỏ. Kích thước này được đo bằng số lượng token, có thể là từ hoặc ký tự. Ví dụ, nếu bạn đặt chunk_size = 50, mỗi đoạn văn bản sẽ chứa 50 token.
- chunk_overlap: Đây là số lượng token sẽ bị trùng lặp giữa các đoạn liên tiếp. Điều này có nghĩa là một phần của đoạn trước sẽ được lặp lại ở đoạn sau. Ví dụ, nếu chunk_overlap = 10, 10 token cuối của một đoạn sẽ được lặp lại làm 10 token đầu của đoạn tiếp theo. Điều này giúp duy trì ngữ cảnh khi chia nhỏ văn bản.
- separator: Đây là ký tự hoặc chuỗi ký tự được sử dụng để phân tách các đoạn văn bản. Ví dụ, bạn có thể sử dụng dấu chấm "." hoặc dấu xuống dòng "\n" để phân chia các đoạn.

Sử dụng TokenTextSplitter, bạn có thể tự động chia nhỏ văn bản thành các đoạn nhỏ hơn mà không cần phải làm thủ công. Điều này giúp tiết kiệm thời gian và đảm bảo các đoạn văn bản được chia một cách hợp lý để mô hình có thể xử lý tốt hơn. Dưới đây là ví dụ:

```

1 from llama_index.core import Document
2 from llama_index.core.node_parser import
3     TokenTextSplitter
4
5 text = "Hôm nay trời nắng, tôi đi ăn kem, lạnh buốt cả răng!"
6 doc = Document(text=text)
7 splitter = TokenTextSplitter(
8     chunk_size=20,
9     chunk_overlap=5,
10    separator= " "
11 )
12 nodes = splitter.get_nodes_from_documents(
13     [doc])
14
15 for node in nodes:
16     print(node)

```

===== Output =====
Metadata length (2) is close to chunk size (20). Resulting chunks are less than 50 tokens. Consider increasing the chunk size or decreasing the size of your metadata to avoid this.
Node ID: 3a186da6-92d1-40e0-823b-0d6f901f426e
Text: Hôm nay trời nắng, tôi đi ăn
Node ID: a075758c-6291-4a5b-975d-8f89850fcfc7
Text: đi ăn kem, lạnh buốt cả răng!
=====

Giải thích đoạn code trên:

- Nhập thư viện: from llama_index.core import Document và from llama_index.core.node_parser import TokenTextSplitter nhập các lớp cần thiết từ thư viện LlamaIndex. Document dùng để tạo tài liệu, còn TokenTextSplitter giúp chia nhỏ văn bản thành các đoạn nhỏ hơn.
- Tạo Document: text = "Hôm nay trời nắng, tôi đi ăn kem, lạnh buốt cả răng!" tạo một chuỗi văn bản đơn giản. doc = Document(text=text) tạo một Document từ chuỗi văn bản trên. Document này sẽ chứa toàn bộ nội dung "Hôm nay trời nắng, tôi đi ăn kem, lạnh buốt cả răng!".

- Tạo TokenTextSplitter: `splitter = TokenTextSplitter(chunk_size=20, chunk_overlap=5, separator="")` tạo một đối tượng TokenTextSplitter với các tham số sau:

- `chunk_size=20`: Mỗi đoạn văn bản sẽ có tối đa 20 token (từ).
- `chunk_overlap=5`: 5 token cuối của một đoạn sẽ được lặp lại ở đầu đoạn kế tiếp.
- `separator=""`: Các đoạn văn bản sẽ được phân chia dựa trên khoảng trắng giữa các từ.

Trong phần output, một cảnh báo xuất hiện nói rằng chúng ta sẽ gặp vấn đề khi mà `chunk_size` của chúng ta quá nhỏ, nhỏ hơn hoặc chênh lệch không quá nhiều so với kích cỡ của metadata. Vì khi đó metadata sẽ chiếm phần lớn lượng dữ liệu của node trong khi dữ liệu thực tế rất ít.

Việc tạo node không chỉ dừng lại ở việc chia nhỏ document, chúng ta có thể thiết lập mối quan hệ giữa các node, ngoài ra còn có một số phương pháp biến đổi node nâng cao hơn mà chúng ta sẽ tìm hiểu ở phần sau. Dưới đây là cách mà chúng ta tạo mối quan hệ giữa hai node, có 5 mối quan hệ giữa các node là: Trong hệ thống phân chia văn bản thành các đoạn nhỏ (nodes) như được sử dụng trong thư viện ‘llama_index’, các mối quan hệ giữa các nodes có thể được mô tả như sau:

- 1. SOURCE:

- Node này là tài liệu gốc (source document). Đây là toàn bộ văn bản ban đầu mà từ đó các nodes khác được tách ra.
- Ví dụ: Nếu văn bản gốc là "Hôm nay trời nắng, tôi đi ăn kem, lạnh buốt cả răng!", thì node ‘SOURCE’ sẽ chứa toàn bộ văn bản này.

- 2. PREVIOUS:

- Node này là node trước đó trong tài liệu. Nó đại diện cho đoạn văn bản ngay trước node hiện tại.
- Ví dụ: Nếu có hai nodes liên tiếp, node thứ hai sẽ có một liên kết ‘PREVIOUS’ trỏ đến node thứ nhất.

- 3. NEXT:

- Node này là node kế tiếp trong tài liệu. Nó đại diện cho đoạn văn bản ngay sau node hiện tại.
- Nếu có hai nodes liên tiếp, node thứ nhất sẽ có một liên kết ‘NEXT’ trỏ đến node thứ hai

- 4. PARENT:

- Node này là node cha trong tài liệu. Nó đại diện cho một đoạn văn bản lớn hơn chứa node hiện tại.
- Ví dụ: Nếu một tài liệu lớn được chia thành các đoạn nhỏ hơn và mỗi đoạn nhỏ này lại được chia thành các đoạn nhỏ hơn nữa, thì một node có thể có một node cha chứa nó.

- 5. CHILD:

- Node này là node con trong tài liệu. Nó đại diện cho một đoạn văn bản nhỏ hơn nằm trong node hiện tại.
- Ví dụ: Nếu một đoạn văn bản lớn được chia thành các đoạn nhỏ hơn, thì mỗi đoạn nhỏ sẽ là một node con của node lớn.

Mặc dù có thể thiết lập các mối quan hệ này một cách thủ công, nhưng mình nhận thấy rằng khi triển khai dự án thực tế, mình chẳng có đủ hơi sức để làm điều này. Dù sao thì khi tạo node thì các mối quan hệ này cũng được tạo tự động rồi, trừ khi mình biết chắc việc tạo thủ công cho kết quả tốt hơn đáng kể thì có thể xem xét lại vấn đề này.

Chúng ta có thể xem các mối quan hệ nào đã được tạo giữa các node trong ví dụ trên bằng cách in ra từng node hoặc đơn giản chỉ in thuộc tính relationships.

```

1 print(nodes[0].relationships)
2 print(nodes[1].relationships)

=====
{<NodeRelationship.SOURCE: '1': RelatedNodeInfo(node_id='7425c6ab-e5c9-4b1b-b12f-2
dcbd2ec946f', node_type=<ObjectType.DOCUMENT: '4'>, metadata={}, hash='0
af2325bc3976ef85024e35769344ac7138fe4c21700c2bf55d2109039882bb4'), <
NodeRelationship.NEXT: '3': RelatedNodeInfo(node_id='a075758c-6291-4a5b-975d-8
f89850fcfc7', node_type=<ObjectType.TEXT: '1'>, metadata={}, hash='524
e34237455ee85fecf2724237a8e825e136e9dff359bbde44c4c3a656c1d95')}
{<NodeRelationship.SOURCE: '1': RelatedNodeInfo(node_id='7425c6ab-e5c9-4b1b-b12f-2
dcbd2ec946f', node_type=<ObjectType.DOCUMENT: '4'>, metadata={}, hash='0
af2325bc3976ef85024e35769344ac7138fe4c21700c2bf55d2109039882bb4'), <
NodeRelationship.PREVIOUS: '2': RelatedNodeInfo(node_id='3a186da6-92d1-40e0-823b-0
d6f901f426e', node_type=<ObjectType.TEXT: '1'>, metadata={}, hash=
edcd2c00a132ff9c5b9c159f7b6bcb26b00e8fc16d6df454d4f1c3639e01c294)}
=====
```

3.4 Indexes

Sau khi bạn nhập dữ liệu vào hệ thống, LlamaIndex sẽ giúp bạn lập chỉ mục dữ liệu vào một cấu trúc dễ dàng truy xuất. Quá trình này thường bao gồm việc tạo ra các vector embeddings và lưu trữ chúng trong một cơ sở dữ liệu chuyên biệt gọi là vector store. Việc lập chỉ mục giúp tổ chức dữ liệu sao cho dễ dàng tìm kiếm và truy xuất sau này. Llama Index hỗ trợ nhiều loại index khác nhau như SummaryIndex, VectorStoreIndex, TreeIndex, KnowledgeGraphIndex, tùy vào từng trường hợp mà chúng ta sẽ chọn loại phù hợp. Nhìn chung, các loại index này đều thực hiện các chức năng tạo index, thêm node mới, và truy vấn. Trong phần này, chúng ta sẽ tìm hiểu cách sử dụng SumaryIndex, theo định nghĩa thì Summary Index là một cấu trúc dữ liệu đơn giản, nơi các Nodes (các đoạn văn bản đã được chia nhỏ) được lưu trữ theo một chuỗi liên tiếp. Đọc tên, có thể chúng ta sẽ nghĩ rằng nó sẽ tóm tắt thông tin các node nhưng mình nghĩ ở đây nó mang nghĩa tổng hợp, vì thực chất, nó chỉ tập hợp lại các node, giống một danh sách trong Python. Phương pháp này rất cơ bản và nó cũng không cả sử dụng đến các mô hình embedding.

Dưới đây là cú pháp sử dụng:

```

1 from llama_index.core import SummaryIndex
2 # Tạo index từ node
3 index = SummaryIndex(nodes)
4 # Tạo index từ document
5 index2 = SummaryIndex.from_documents([doc])
```

Trên đây là 2 cách để chúng ta tạo index, cách 1 tạo index từ node, cách 2 tạo index từ document. Điểm khác biệt là khi tạo index từ node, chúng ta sẽ cần thiết lập thủ công các tham số về kích thước mỗi node,... trong khi cách 2 tạo index từ document, quá trình tạo node diễn ra hoàn toàn tự động.

Ví dụ sau đây sẽ thực hiện tạo index theo cả hai cách:

```

1 from llama_index.core import Document, SummaryIndex
2 from llama_index.core.schema import TextNode
3 from llama_index.core.node_parser import TokenTextSplitter
4
5 text = "Con mèo ú nầm ướn bên cửa sổ. Tôi muốn mình cũng được như thế."
6 doc = Document(
7     text=text
8 )
9
10 splitter = TokenTextSplitter(
11     chunk_size=20,
12     chunk_overlap=5,
13     separator=" "
14 )
15 nodes = splitter.get_nodes_from_documents([doc])
16 index = SummaryIndex(nodes)
17 index2 = SummaryIndex.from_documents([doc])
18
19 print("index 1", index.index_struct)
20 print("index 2", index2.index_struct)

```

```

=====
index 1 IndexList(index_id='fde51551-b82f-4be0-a7c7-af11cbd790d1', summary=None, nodes
=[ '37d4af84-6deb-4b5a-8906-73eff559ce10', '9fa671af-9be6-43ea-8fd0-5d603c6cc0c0',
  '0b75eed6-fd63-4b7a-bbd2-b8ed9962a3c6'])
index 2 IndexList(index_id='ab9305a7-9268-4ad1-8edf-368aef1d8c8d', summary=None, nodes
=[ '796e0d8c-8e7e-4ee2-91aa-a74df80207a6'])
=====
```

Giải thích code:

- Đầu tiên chúng ta cần khai báo các gói cần thiết, trong đó SumaryIndex nằm trong llama_index.core
- Tiếp theo chúng ta tạo document và nodes như cách mà chúng ta đã tìm hiểu ở hai phần trước đó.
- Việc tạo index thực hiện theo hai phương pháp, đầu tiên index sẽ được tạo bằng cách sử dụng các Nodes đã chia nhỏ. Các Nodes này được lưu trữ theo trình tự trong index. Trong khi đó cách 2 index được tạo trực tiếp từ Document, tức là văn bản ban đầu sẽ tự động được chia nhỏ và lưu trữ dưới dạng Nodes trong index2.
- Chúng ta có thể xem cấu trúc của các index đã tạo qua thuộc tính index_struct của nó.

3.5 Truy vấn dữ liệu

Trong phần trước, chúng ta đã tạo ra một SummaryIndex từ các Nodes bằng cách sử dụng SummaryIndex. Bạn có thể xem cấu trúc của Index thông qua thuộc tính index_struct. Đến đây, Index đã sẵn sàng để truy xuất thông tin. Nhìn chung, các công đoạn từ việc tạo Document, chia nhỏ thành Nodes, rồi tạo Index đều nhằm một mục tiêu chính: truy xuất thông tin từ dữ liệu. Để làm điều này, LlamaIndex cung cấp nhiều công cụ khác nhau, nhưng cách đơn giản nhất là tạo một đối tượng query_engine từ index bằng phương thức index.as_query_engine().

Nói cách khác, chúng ta sẽ tạo ra một công cụ hỏi đáp từ Index đã tạo. Để sử dụng công cụ này, bạn chỉ cần gọi phương thức query_engine.query() và truyền vào câu hỏi mà bạn muốn kiểm thông tin.

```

1 query_engine = index.as_query_engine()
2 responce = query_engine.query("mèo ú đang l
  àm gì?")
3 print(responce)
4
5
6
7
8 \***\

```

=====
Output =====
ValueError: No API key found for OpenAI.
Please set either the OPENAI_API_KEY
environment variable or openai.api_key
prior to initialization.
API keys can be found or created at https
://platform.openai.com/account/api-
keys
=====

Bùm, và lỗi xuất hiện, lỗi này là do trình truy xuất dữ liệu của chúng ta cần sử dụng đến API của OPEN AI. Để sửa lỗi thì chúng ta cần thiết lập khóa API này. Để thiết lập có nhiều cách, chúng ta sẽ làm theo cách dưới đây:

```

1 from llama_index.core import Settings
2 from llama_index.llms.openai import OpenAI
3 import openai
4
5 openai.api_key = "sk-your-openai-api-key"
6 Settings.llm = OpenAI(model="gpt-4o-mini", temperature=0.2)

```

Đoạn mã trên thực hiện hai nhiệm vụ:

- Đầu tiên là thiết lập API key: Đây là bước đầu tiên và cần thiết để kết nối với OpenAI API, cho phép chúng ta sử dụng các dịch vụ của OpenAI như các mô hình ngôn ngữ lớn, các mô hình embedding.
- Tiếp theo chúng ta cấu hình mô hình ngôn ngữ lớn chúng ta sẽ sử dụng:
 - Settings.llm: Đây là thuộc tính giúp chúng ta thiết lập mô hình ngôn ngữ mà LlamaIndex sẽ sử dụng cho việc truy vấn thông tin.
 - OpenAI(model="gpt-4o-mini", temperature=0.2): Trong đoạn mã này, chúng ta chọn sử dụng mô hình "gpt-4o-mini" từ OpenAI với tham số temperature=0.2.
 - * model: Mô hình mà bạn muốn sử dụng, ví dụ như "gpt-4o-mini". Đây là phiên bản nhỏ hơn và nhẹ hơn của mô hình GPT-4.
 - * temperature: Tham số này có giá trị từ 0 đến 2, nó kiểm soát sự sáng tạo của mô hình khi đưa ra câu trả lời. Giá trị thấp hơn (như 0.2) sẽ khiến mô hình trả lời cùi chỏ và ít ngẫu nhiên hơn, giúp kết quả trở nên chính xác hơn.

Sau khi hoàn tất các thiết lập, hệ thống RAG đơn giản của chúng ta đã sẵn sàng để sử dụng.

```

1 query_engine = index.as_query_engine()
2 responce = query_engine.query("mèo ú đang
  làm gì?")
3 print(responce)

```

=====
Output =====
Mèo ú đang nằm ườn bên cửa sổ.
=====

Ở đây, QueryEngine hoạt động như sau:

- Đầu tiên trình truy xuất sẽ tìm kiếm và lấy ra các phần (Nodes) liên quan từ chỉ mục dựa trên câu hỏi của bạn.
- Sau đó các phần này được sắp xếp lại hoặc lọc để đảm bảo thông tin cần thiết được chuẩn bị tốt nhất.

- Cuối cùng, nó kết hợp các phần đó và đưa vào mô hình ngôn ngữ (LLM) để tạo ra câu trả lời hoàn chỉnh cho bạn.

Tóm tắt

Vậy là chúng ta đã tìm hiểu về các khái niệm cơ bản trong llama index và xây dựng thành công mô hình truy vấn RAG đơn giản. Dưới đây là tóm tắt các bước xây dựng mô hình RAG cơ bản:

- 1. Tải dữ liệu dưới dạng Documents
- 2. Phân tích tài liệu thành các Node
- 3. Xây dựng chỉ mục được tối ưu hóa từ Node
- 4. Truy vấn

4 Kỹ thuật Ingesting Data

Code phần này có tại: [docs/tutorial/2.Ingesting-data.ipynb](#)

4.1 Loading Data

Một câu trả lời tốt đến từ dữ liệu chất lượng, vì vậy việc chuẩn bị và xử lý dữ liệu cho hệ thống RAG rất quan trọng. Trong phần Llamaindex cơ bản, chúng ta đã nắm được khái niệm về Node, Document, Index. Việc tạo Documents trong các ví dụ trước đó sử dụng đầu vào là string đơn giản. Tuy nhiên trong thực tế, chúng ta có rất nhiều loại dữ liệu khác nhau, vậy làm thế nào để đọc được chúng.

Llamaindex cung cấp cho chúng ta nhiều công cụ có sẵn để giải quyết vấn đề này, trong dự án này sẽ sử dụng công cụ SimpleDirectoryReader, ngoài ra chúng ta còn học cách sử dụng thêm một số công cụ khác từ kho thư viện Llamahub.

a) SimpleDirectoryReader

SimpleDirectoryReader cực kỳ tiện dụng bởi khả năng đọc được hầu hết các loại tài liệu phổ biến như PDF, Word, CSV, TXT. Việc của chúng ta là tạo một thư mục chứa các loại dữ liệu và đưa cho SimpleDirectoryReader, nó sẽ tự xử lý, nó tự động đến mức mà chúng ta không cần phải chỉ rõ từng loại file, nó tự động lựa chọn công cụ phù hợp với từng file mà nó nhận dạng bởi đuôi file. Với các file PDF, CSV nó sử dụng các thư viện như PyPDF... để tải các dữ liệu này.

Cú pháp:

```
1 from llama_index.core import SimpleDirectoryReader
2 documents = SimpleDirectoryReader(input_dir: str, input_files: List, recursive: bool)
```

Trong đó:

- input_dir: str: Đường dẫn đến thư mục chứa các tệp tin mà bạn muốn nạp dữ liệu từ đó.
- input_files: List: Danh sách các tên tệp cụ thể, mặc định là None.
- recursive: bool: Nếu đặt True, SimpleDirectoryReader sẽ tìm kiếm và nạp dữ liệu từ tất cả các thư mục con bên trong input_dir. Nếu đặt False, chỉ các tệp trong thư mục chính mới được nạp.

Ví dụ dưới đây sẽ sử dụng SimpleDirectoryReader để đọc file trong một thư mục chứa nhiều file:

```
1 from llama_index.core import SimpleDirectoryReader
2
3 STORAGE_PATH = "docx"
4 documents = SimpleDirectoryReader(
5     STORAGE_PATH,
6     recursive=True
7 ).load_data()
8 for i in documents[0]:
9     print(i)
```

Kết quả trả về là danh sách các đối tượng Document được tạo từ các tài liệu, mỗi đối tượng có các thuộc tính như id, metadata, text...

b) SimpleWebPageReader

SimpleWebPageReader là một công cụ giúp tải nội dung từ các trang web trực tiếp vào LlamaIndex. Công cụ này tự động chuyển đổi HTML thành văn bản để dễ dàng xử lý và phân tích.

Cú Pháp:

```
1 reader = SimpleWebPageReader(html_to_text=True)
2 pages = reader.load_data(urls=["URL1", "URL2", ...])
```

Trong đó:

- html_to_text: bool : Nếu đặt ‘True‘, nội dung HTML sẽ được chuyển đổi thành văn bản thuận tiện.
- urls: List[str]: Danh sách các URL của các trang web mà ta muốn tải nội dung.

Ví dụ dưới đây chúng ta sẽ tải nội dung một bài viết trên trang web suckhoevadoisong.vn:

```
1 from llama_index.readers.web import SimpleWebPageReader
2
3 reader = SimpleWebPageReader(html_to_text=True)
4 pages = reader.load_data(urls=["https://suckhoevadoisong.vn/viet-nam-co-khoang-14-trieu-
5     nguoi-roi-loan-tam-than-169230803214300404.htm"])
5 print(pages[0].text)
```

Dưới đây là phần giải thích đoạn mã trên:

- SimpleWebPageReader được khởi tạo với html_to_text=True, nghĩa là trang web sẽ được chuyển đổi từ HTML sang văn bản.
- load_data được sử dụng để tải nội dung từ URL được chỉ định. Trong ví dụ này, nội dung từ trang báo sức khỏe sẽ được tải về.
- print(pages[0].text) sẽ in ra văn bản đã được chuyển đổi từ trang web đó.

c) LlamaParse

LlamaParse là nền tảng phân tích tài liệu genAI đầu tiên trên thế giới, được xây dựng đặc biệt cho các trường hợp sử dụng liên quan đến LLMs (Large Language Models). Mục tiêu chính của LlamaParse là giúp phân tích và làm sạch dữ liệu để đảm bảo chất lượng tốt nhất trước khi sử dụng trong các ứng dụng liên quan đến LLMs, như Retrieval-Augmented Generation (RAG).

Chức Năng Chính:

- Trích xuất bảng hiện đại: LlamaParse có khả năng trích xuất dữ liệu từ các bảng một cách hiệu quả.
- Hỗ trợ nhiều loại tệp: Hỗ trợ hơn 10 loại tệp khác nhau như .pdf, .pptx, .docx, .html, .xml, và nhiều hơn nữa.
- Hỗ trợ ngôn ngữ nước ngoài: Có khả năng làm việc với nhiều ngôn ngữ khác nhau.

Ví Dụ: Ví dụ sau đây, chúng ta sử dụng LlamaParse để phân tích một tài liệu PDF. Chúng ta thiết lập môi trường, tạo một đối tượng parser với result_type="text" để chỉ định rằng chúng ta muốn đầu ra là văn bản. Sau đó, chúng ta sử dụng SimpleDirectoryReader để đọc dữ liệu từ thư mục "files/" và phân tích nội dung của các tệp PDF bằng LlamaParse.

```

1 from llama_parse import LlamaParse
2 from llama_index.core import SimpleDirectoryReader
3 import os
4
5 # Thêm key vào môi trường
6 os.environ["LLAMA_CLOUD_API_KEY"] = "llx-your-llama-cloud-api-key"
7
8 # Khởi tạo đối tượng LlamaParse
9 parser = LlamaParse(result_type="text")
10
11 # Chỉ định trình phân tích tệp cho .pdf
12 file_extractor = {".pdf": parser}
13
14 # Đọc dữ liệu từ thư mục "files/"
15 reader = SimpleDirectoryReader(
16     "files/",
17     file_extractor=file_extractor
18 )
19 docs = reader.load_data()

```

Để thực hiện ví dụ trên, chúng ta cần tạo Llama Cloud API Key tại trang web cloud.llamaindex.ai. Hiện tại Llamacloud cho phép chúng ta sử dụng miễn phí với giới hạn sử dụng xử lí 1000 trang tài liệu một ngày. Mình đánh giá rất cao công cụ này bởi khả năng xử lí các định dạng tài liệu mạnh mẽ. Trong ví dụ trên, file pdf có cấu trúc phức tạp gồm cả văn bản và bảng, nhưng LlamaParse cho kết quả đọc file rất xuất sắc, các bạn có thể so sánh với việc đọc file chỉ dùng SimpleDirectoryReader thông thường để thấy sự khác biệt này. Nhưng nói chung không có miếng bánh nào miễn phí cả, đối với các ứng dụng thương mại, chúng ta cần phát triển một công cụ riêng sẽ tối ưu hơn.

4.2 Các kỹ thuật tạo node

Trong phần Llamaindex cơ bản, chúng ta đã được học về phương pháp tạo node với TokenTextSplitter. Một phương pháp mà chúng ta tạo node bằng cách chia nhỏ Document thành các phần với kích thước mà chúng ta chỉ định. Trong phần này, chúng ta sẽ tìm hiểu thêm cách sử dụng các công cụ khác.

a) SimpleFileNodeParser

SimpleFileNodeParser là một công cụ trong LlamaIndex giúp phân tích tài liệu từ các tệp và tự động chọn phương pháp phân tích nút (node) tốt nhất cho từng loại nội dung. Nó đặc biệt hữu ích khi làm việc với nhiều loại tệp khác nhau.

Trong ví dụ dưới đây, chúng ta sử dụng FlatReader để tải dữ liệu từ tệp Markdown (.md), sau đó sử dụng SimpleFileNodeParser để phân tích dữ liệu và tạo ra các nút từ tài liệu.

```

1 from llama_index.core.node_parser import SimpleFileNodeParser
2 from llama_index.readers.file import FlatReader
3 from pathlib import Path
4
5 md_docs = FlatReader().load_data(Path("files/md_test.md"))
6
7 parser = SimpleFileNodeParser()
8 md_nodes = parser.get_nodes_from_documents(md_docs)
9 print(md_nodes[0])

```

```

=====
Output =====
Node ID: 87571ec5-00ab-40cc-b2df-63edcc34309d
Text: Documents Một thành phần không thể thiếu trong các ứng dụng RAG
là dữ liệu, chúng ta có nhiều loại dữ liệu khác nhau như: pdf, docx, ...
=====
```

Ngoài ra chúng ta có thể sử dụng các trình phân tích cú pháp cho từng loại dữ liệu cụ thể như: HTMLNodeParser, JSONNodeParser, MarkdownNodeParser. Nhưng mình cho rằng chỉ cần dùng SimpleFileNodeParser là đủ, nó sẽ lựa chọn các công cụ phù hợp nhất với dữ liệu.

b) Các bộ chia văn bản - Splitters

Chúng ta vừa tìm hiểu về công cụ SimpleFileNodeParser có thể tạo node từ nhiều loại dữ liệu khác nhau. Nhưng chúng ta không có lựa chọn nào để điều chỉnh các thông số như kích cỡ node... Vậy nên, trong trường hợp muốn điều chỉnh thông số hay muốn tùy chỉnh việc tạo node. Chúng ta sẽ sử dụng các công cụ thuộc nhóm Splitter. Có khá nhiều công cụ mà LlamaIndex cung cấp như:

- CodeSplitter: Tạo node từ việc phân tách dữ liệu code. Ví dụ như các chương trình python.
- SentenceSplitter: phân tách văn bản trong khi vẫn tôn trọng ranh giới của câu.
- SentenceWindowNodeParser: Mỗi node được tạo từ 1 câu, ngoài ra nó còn sử dụng các câu phía trước và phía sau câu hiện tại để làm ngữ cảnh.
- SemanticSplitterNodeParser: Thay vì tạo node từ các câu theo độ dài cố định, nó lấy các câu theo ngữ nghĩa.
- TokenTextSplitter: Phương pháp tạo node bằng cách chia Document thành những phần nhỏ hơn theo chunk_size mà chúng ta đã được giới thiệu trong phần trước.
- HierarchicalNodeParser: Thực hiện tạo node theo phương pháp phân cấp.

Trong các phương pháp trên, tùy trường hợp mà chúng ta sẽ lựa chọn phương pháp phù hợp, cách ví dụ cho từng phương pháp các bạn xem thêm ở [node_parsers](#). Phương pháp SemanticSplitterNodeParser khá hay nên chúng ta hãy cùng tìm hiểu phương pháp này.

"Semantic Chunking" là một phương pháp mới được đề xuất bởi Greg Kamradt, thay vì chia văn bản theo kích thước cố định, phương pháp này sử dụng sự tương đồng ngữ nghĩa để chọn điểm ngắt giữa các câu. Điều này đảm bảo rằng mỗi "chunk" chứa các câu có liên quan về mặt ngữ nghĩa, giúp tạo ra các khối văn bản có ý nghĩa hơn khi làm việc với các mô hình ngôn ngữ lớn (LLMs).

Phương pháp này đã được tích hợp vào module LlamaIndex với tên gọi SemanticSplitterNodeParser. Trong ví dụ dưới đây, chúng ta sử dụng SemanticSplitterNodeParser để phân chia văn bản dựa trên ngữ nghĩa, thay vì sử dụng kích thước cố định. Điều này được thực hiện bằng cách tính toán độ tương đồng giữa các câu và chọn điểm ngắt phù hợp.

```

1 from llama_index.core.node_parser import SemanticSplitterNodeParser
2 from llama_index.embeddings.openai import OpenAIEmbedding
3 from llama_index.core import Document
4
5 text = """Mèo rất đáng yêu.
6     Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.
7     Ở dạng đậm đặc, acid này có thể tạo thành các sương mù acid,
8     chúng đều có khả năng ăn mòn các mô con người, gây tổn thương cơ quan hô hấp,
9     mắt, da và ruột. """
10
11 doc = Document(text=text)
12
13 # Sử dụng mô hình OpenAI để tính toán embedding
14 embed_model = OpenAIEmbedding()
15
16 # Khởi tạo SemanticSplitterNodeParser với các tham số tùy chỉnh
17 splitter = SemanticSplitterNodeParser(
18     buffer_size=1,
19     breakpoint_percentile_threshold=95,
20     embed_model=embed_model
21 )
22
23 nodes = splitter.get_nodes_from_documents([doc])
24 for node in nodes:
25     print(node)

```

```

=====
Output =====
Node ID: 6d15126c-007d-4dc2-9123-7287da01afc1
Text: Mèo rất đáng yêu.
Node ID: 5971de1f-8369-4fc6-b7ec-478a5697bb54
Text: Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.
Ở dạng đậm đặc, acid này có thể tạo thành các sương mù acid,
chúng đều có khả năng ăn mòn các mô con người, gây tổn thương cơ quan
hô hấp, mắt, da và ruột.
=====
```

Trong ví dụ trên:

- buffer_size=1: Tham số này xác định số lượng câu được xem xét trước và sau câu hiện tại để xác định điểm ngắt. Với buffer_size=1, splitter sẽ xem xét câu trước và câu sau câu hiện tại khi đánh giá mức độ tương đồng ngữ nghĩa.
- breakpoint_percentile_threshold=95: Đây là ngưỡng phần trăm được sử dụng để xác định điểm ngắt giữa các câu. Nếu điểm tương đồng ngữ nghĩa giữa hai câu thấp hơn ngưỡng 95%, thì đó là một điểm ngắt tiềm năng. Ngưỡng càng cao, việc ngắt câu sẽ càng ít xảy ra, và các chunk sẽ lớn hơn.
- embed_model=embed_model: Đây là mô hình embedding được sử dụng để tính toán mức độ tương đồng ngữ nghĩa giữa các câu. Ở đây, chúng ta sử dụng mô hình embedding của OpenAI để tạo ra các vector đại diện cho từng câu, giúp xác định mức độ liên quan giữa các câu trong văn bản.

Kết quả chương trình đã tạo cho chúng ta 2 node, mỗi node chứa nội dung riêng biệt theo ngữ nghĩa câu chúng tôi phương pháp này rất tiềm năng. Tuy nhiên việc sử dụng embedding từ API của OPENAI sẽ tốn phí. Chúng ta có thể thay thế bằng các mô hình tương đương trên huggingface.

4.3 Khai thác sức mạnh của metadata - Metadata Extraction

Metadata Extractor là công cụ giúp cải thiện quá trình truy xuất thông tin từ các đoạn văn bản, đặc biệt khi chúng có nội dung tương tự nhau. Bằng cách sử dụng LLMs để trích xuất thông tin ngữ cảnh liên quan, Metadata Extractor giúp mô hình nhận diện và phân biệt các đoạn văn bản hiệu quả hơn. Dưới đây là các công cụ trong Metadata Extractor:

- TitleExtractor(): Trích xuất tiêu đề hoặc nội dung chính của tài liệu để dễ dàng định vị thông tin quan trọng.
- QuestionsAnsweredExtractor(): Trích xuất các câu hỏi mà đoạn văn bản trả lời, giúp xác định nhanh nội dung chính.
- SummaryExtractor(): Tạo tóm tắt cho đoạn văn bản hiện tại và cả các đoạn trước đó, giúp duy trì ngữ cảnh.
- KeywordExtractor(): Trích xuất các từ khóa quan trọng từ văn bản để định vị nội dung cốt lõi.
- EntityExtractor(): Trích xuất các thực thể (như tên người, địa điểm) với độ chính xác cao, giúp định danh thông tin quan trọng trong đoạn văn bản.

Trước khi đi vào phần hướng dẫn sử dụng từng công cụ, chúng ta cùng xem lại cấu trúc của Document, Node.

```

1 from llama_index.core.node_parser import SemanticSplitterNodeParser
2 from llama_index.embeddings.openai import OpenAIEmbedding
3 from llama_index.core import SimpleDirectoryReader
4
5 # Đọc dữ liệu từ thư mục
6 reader = SimpleDirectoryReader(input_files=["files/test_metadata_ex.txt"])
7 docs = reader.load_data()
8
9 # Sử dụng mô hình OpenAI để tính toán embedding
10 embed_model = OpenAIEmbedding()
11
12 # Khởi tạo SemanticSplitterNodeParser với các tham số tùy chỉnh
13 splitter = SemanticSplitterNodeParser(
14     buffer_size=1,
15     breakpoint_percentile_threshold=95,
16     embed_model=embed_model
17 )
18
19 nodes = splitter.get_nodes_from_documents(docs)
20
21 print("document metadata", docs[0].metadata)
22 print("node metadata", nodes[0].metadata)

=====
document metadata {'file_path': 'files/test_metadata_ex.txt', 'file_name': 'test_metadata_ex.txt', 'file_type': 'text/plain', 'file_size': 308, 'creation_date': '2024-08-12', 'last_modified_date': '2024-08-12'}

node metadata {'file_path': 'files/test_metadata_ex.txt', 'file_name': 'test_metadata_ex.txt', 'file_type': 'text/plain', 'file_size': 308, 'creation_date': '2024-08-12', 'last_modified_date': '2024-08-12'}
=====
```

Thành phần metadata trong Document được tạo tự động, chứa thông tin về file dữ liệu gốc như vị trí.... Khi tạo node, dữ liệu này tự động được gán cho node nên chúng ta có thể thấy kết quả đầu ra metadata của node con giống hệt trong document.

Quay trở lại với nội dung chính của phần này, nhiệm vụ của chúng ta là khai thác thuộc tính metadata trong node để hỗ trợ việc tìm kiếm hiệu quả hơn. Theo như kết quả của ví dụ trên thì chúng ta chưa thấy được lợi ích nào của metadata trong node. Chính vì vậy, chúng ta sẽ sử dụng các phương pháp trích xuất metadata từ node để tạo ra những thông tin có giá trị.

a) TitleExtractor

TitleExtractor là công cụ giúp chúng ta trích xuất tiêu đề cho các node. Nó giúp chúng ta phân loại các node theo các chủ đề khác nhau, từ đó nâng cao hiệu quả khi tìm kiếm. Việc này giống như chúng ta có một cuốn sách dày cộp, chúng ta muốn tìm kiếm nội dung thuộc một chủ đề cụ thể, nhưng các nội dung này nằm rải rác khắp cuốn sách, nếu đọc hết cuốn sách để tìm từng đoạn thì mất rất nhiều công sức phải không nào? Trong trường hợp này, nếu sử dụng TitleExtractor thì sẽ hiệu quả hơn nhiều. Hãy cùng xem ví dụ dưới đây:

```

1 import nest_asyncio
2 nest_asyncio.apply()
3 from llama_index.core.extractors import TitleExtractor
4 title_extractor = TitleExtractor()
5 metadata_list = title_extractor.extract(nodes)
6 for metadata in metadata_list:
7     print(metadata)

=====
===== Output =====
{'document_title': '"Khám Phá Thế Giới Động Vật và Hóa Học: Từ Sự Đáng Yêu Của Mèo Đến
Tác Động Của Axit Hydrochloric"'}
{'document_title': '"Khám Phá Thế Giới Động Vật và Hóa Học: Từ Sự Đáng Yêu Của Mèo Đến
Tác Động Của Axit Hydrochloric"'}
=====
```

Trong ví dụ trên chúng ta thực hiện trích xuất title cho các node, các tham số trong TitleExtractor chúng ta sử dụng mặc định. Tuy nhiên, có một số tham số có thể điều chỉnh là:

- llm: Loại llm chúng ta sử dụng
- nodes: Số lượng node sử dụng để tóm tắt, mặc định là 5
- node_template: Prompt dùng trích xuất tiêu đề, mặc định dùng mẫu có sẵn của thư viện.

Đây là nội dung của title template mặc định, chúng ta hoàn toàn có thể dựa vào đây để thiết kế một mẫu mới:

```

1 DEFAULT_TITLE_NODE_TEMPLATE = """\
2 Context: {context_str}. Give a title that summarizes all of \
3 the unique entities, titles or themes found in the context. Title: """
```

b) QuestionsAnsweredExtractor

QuestionsAnsweredExtractor giúp xác định các câu hỏi mà văn bản có thể trả lời, giúp quá trình truy xuất tập trung vào những nút có thông tin chính xác cho các câu hỏi cụ thể. Ví dụ, trong hệ thống FAQ, công cụ này giúp nhận diện các câu hỏi duy nhất mà bài viết có thể trả lời, từ đó dễ dàng tìm kiếm câu trả lời chính xác cho các truy vấn của người dùng.

Ví dụ:

```
1 from llama_index.core.extractors import QuestionsAnsweredExtractor
2 qa_extractor = QuestionsAnsweredExtractor(questions=3)
3 metadata_list = qa_extractor.extract(nodes)
4 print(metadata_list)
```

```
=====
[{'questions_this_excerpt_can_answer': 'Based on the provided context, here are three specific questions that can be answered using the information given:\n\n1. **What is the file size of the document named "test_metadata_ex.txt"?**\n - Answer: The file size is 308 bytes.\n\n2. **On what date was the file "test_metadata_ex.txt" created?**\n - Answer: The file was created on August 12, 2024.\n\n3. **What is the MIME type of the file "test_metadata_ex.txt"?**\n - Answer: The file type is text/plain.'}, {'questions_this_excerpt_can_answer': "Based on the provided context about hydrochloric acid and its properties, here are three specific questions that can be answered:\n\n1. **What is the maximum concentration of concentrated hydrochloric acid?**\n...\n"}]
```

Theo kinh nghiệm của mình, với các dự án sử dụng ngôn ngữ khác như Tiếng Việt thì chúng ta cần thiết kế, hay đơn giản là chỉnh sửa Propmt mặc định sang tiếng việt. Ví dụ như trường hợp trên, kết quả trả về là tiếng Anh, mình nghĩ nó sẽ có vấn đề mặc dù các mô hình LLM hiện tại hỗ trợ đa ngôn ngữ.

Sau đây, chúng ta sẽ chỉnh sửa prompt để kết quả đầu ra là tiếng việt. Chúng ta sẽ làm theo cách đơn giản nhất là lấy propmt mặc định rồi viết thêm yêu cầu trả về kết quả tiếng Việt là xong.

```
1 CUSTORM_QUESTION_GEN_TMPL = """\
2 Here is the context:
3 {context_str}
4
5 Given the contextual information, \
6 generate {num_questions} questions this context can provide \
7 specific answers to which are unlikely to be found elsewhere.
8
9 Higher-level summaries of surrounding context may be provided \
10 as well. Try using these summaries to generate better questions \
11 that this context can answer.
12
13 Lưu ý: Hãy trả về kết quả bằng tiếng Việt.
14 """
```

Đối với chương trình thì vẫn giữ như cũ, chỉ thêm tham số prompt_template khi khởi tạo QuestionsAnsweredExtractor.

```
1 from llama_index.core.extractors import QuestionsAnsweredExtractor
2 qa_extractor = QuestionsAnsweredExtractor(questions=3, prompt_template=
3     CUSTORM_QUESTION_GEN_TMPL)
4 metadata_list = qa_extractor.extract(nodes)
4 print(metadata_list)
```

Và kết quả đầu ra sẽ là tiếng Việt.

```
===== Output =====
[{'questions_this_excerpt_can_answer': '1. Tệp tin "test_metadata_ex.txt" được tạo ra
vào ngày nào?\n2. Kích thước của tệp tin "test_metadata_ex.txt" là bao nhiêu byte?\n
3. Nội dung của tệp tin "test_metadata_ex.txt" có chứa thông tin gì đặc biệt không
?'}, {'questions_this_excerpt_can_answer': '1. Nồng độ tối đa của acid hydrochloric
ở dạng đậm đặc là bao nhiêu phần trăm?\n2. Acid hydrochloric có thể gây tổn thư
ng cho những bộ phận nào của cơ thể con người?\n3. Ngày nào acid hydrochloric đượ
c ghi nhận trong tài liệu này?'}]
=====
```

c) SummaryExtractor

SummaryExtractor tạo ra các tóm tắt ngắn gọn cho từng nút và các nút lân cận, giúp tăng cường hiệu quả truy xuất thông tin.

```
1 from llama_index.core.extractors import SummaryExtractor
2
3 CUSTOM_SUMMARY_EXTRACT_TEMPLATE = """\
4 Here is the content of the section:
5 {context_str}
6
7 Summarize the key topics and entities of the section. \
8 Lưu ý: Hãy trả về kết quả bằng tiếng Việt.\n
9 Summary: """
10 summary_extractor = SummaryExtractor(summaries=["prev", "self", "next"],
11                                     prompt_template=CUSTOM_SUMMARY_EXTRACT_TEMPLATE)
12 metadata_list = summary_extractor.extract(nodes)
13 print(metadata_list)
14
===== Output =====
[{'next_section_summary': 'Tóm tắt:\n\n- **Chủ đề chính**: Acid hydrochloric (HCl) và
tác động của nó.\n- **Nồng độ**: Acid hydrochloric đậm đặc nhất có nồng độ tối đa
là 40%.\n- **Tác hại**: Acid hydrochloric có thể tạo thành sương mù acid, gây ăn mòn
mô con người và tổn thương cho các cơ quan như hô hấp, mắt, da và ruột.', 'section_summary': 'Tóm tắt: \n\nNội dung của phần này chứa thông tin về một tệp tin
có tên "test_metadata_ex.txt". Tệp này thuộc loại văn bản (text/plain),
có kích thước 308 byte, và được tạo ra cũng như chỉnh sửa vào ngày 12 tháng 8 năm
2024.'},
{'prev_section_summary': 'Tóm tắt: \n\nNội dung của phần này chứa thông tin về một tệp tin
có tên "test_metadata_ex.txt". Tệp này thuộc loại văn bản (text/plain), có kích thước 308 byte, và
được tạo ra cũng như chỉnh sửa vào ngày 12 tháng 8 năm 2024. Ngoài ra, phần nội dung của tệp tin
đề cập đến mèo, mô tả chúng là rất đáng yêu.\n\nCác chủ đề và thực thể chính bao gồm:\n- Tên tệp: test_metadata_ex.txt\n- Loại tệp
: văn bản\n- Kích thước tệp: 308 byte\n- Ngày tạo và chỉnh sửa: 12/08/2024\n- Nội
dung: Mèo đáng yêu.', 'next_section_summary': 'Tóm tắt: \n\nNội dung của phần này
chứa thông tin về một tệp tin có tên "test_metadata_ex.txt".\n\nTệp này thuộc loại văn bản (text/plain), có kích thước 308 byte, và
được tạo ra cũng như chỉnh sửa vào ngày 12 tháng 8 năm 2024. Ngoài ra, phần nội dung của tệp tin
đề cập đến mèo, mô tả chúng là rất đáng yêu.\n\nCác chủ đề và thực thể chính bao gồm
:\n- Tên tệp: test_metadata_ex.txt\n- Loại tệp: văn bản\n- Kích thước tệp: 308 byte
\n- Ngày tạo và chỉnh sửa: 12/08/2024\n- Nội dung: Mèo đáng yêu.', 'section_summary': 'Tóm tắt:\n\n- **Chủ đề chính**: Acid hydrochloric (HCl) và tác động của nó.\n- **Nồng độ**: Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.\n- **Tác hại**: Acid hydrochloric có thể tạo thành sương mù acid, gây ăn mòn mô con người và
tổn thương cho các cơ quan như hô hấp, mắt, da và ruột.']}
=====
```

Trong ví dụ trên chúng ta cần chú ý đến tham số summaries: Danh sách các loại tóm tắt mà ta muốn tạo cho các nút. Các giá trị có thể là:

- "self": Tóm tắt nội dung của nút hiện tại.
- "prev": Tóm tắt nội dung của nút trước đó.
- "next": Tóm tắt nội dung của nút tiếp theo.

d) KeywordExtractor

KeywordExtractor giúp trích xuất các từ khóa quan trọng từ nội dung văn bản. Những từ khóa này rất hữu ích trong việc tìm kiếm và truy xuất các nút liên quan dựa trên truy vấn của người dùng

```

1 from llama_index.core.extractors import KeywordExtractor
2
3 key_extractor = KeywordExtractor(keywords=3)
4 metadata_list = key_extractor.extract(nodes)
5 print(metadata_list)

=====
===== Output =====
[{'excerpt_keywords': 'Keywords: mèo, đáng yêu, thú cưng'}, {'excerpt_keywords': 'Keywords: acid hydrochloric, nồng độ, ăn mòn'}]
=====
```

Trong ví dụ trên, chúng ta tiến hành trích xuất 3 keyword từ mỗi node, các tham số tương tự như các trình trích xuất khác.

e) EntityExtractor

EntityExtractor giúp nhận diện và trích xuất các thực thể được đặt tên từ văn bản, chẳng hạn như tên người, địa điểm, tổ chức, và nhiều loại thực thể khác. Việc này giúp hệ thống truy xuất có thể tập trung vào các nút có chứa thông tin cụ thể, cung cấp ngữ cảnh cần thiết để cải thiện độ chính xác của kết quả tìm kiếm.

Đối với công cụ này, yêu cầu chúng ta cài đặt thêm gói bổ xung sau:

```

1 pip install llama-index-extractors-entity
2 pip install transformers==4.40.2
```

Tiếp theo chúng ta sẽ trích xuất tên các thực thể :

```

1 import nest_asyncio
2 nest_asyncio.apply()
3 from llama_index.extractors.entity import EntityExtractor
4
5
6 entity_extractor = EntityExtractor(
7     # label_entities=True,
8     device="cpu"
9 )
10 metadata_list = entity_extractor.extract(nodes)
11 print(metadata_list)

=====
===== Output =====
[{}, {'entities': ['hô hấp']}]
=====
```

Trong chương trình trên, có một số tham số chúng ta cần chú ý:

- label_entities: Nếu đặt thành True, trình trích xuất sẽ gắn nhãn (label) cho mỗi tên thực thể với một loại thực thể tương ứng, ví dụ như người, địa điểm, hoặc tổ chức. Điều này rất hữu ích trong giai đoạn truy xuất và truy vấn sau này. Mặc định, tham số này được đặt thành False.

- device: Xác định thiết bị mà mô hình sẽ chạy trên đó. Mặc định là "cpu", nhưng nếu hệ thống của chúng ta hỗ trợ, ta có thể đặt thành "cuda" để sử dụng GPU, giúp tăng tốc quá trình xử lý.
- entity_map: Cho phép ta tùy chỉnh nhãn cho từng loại thực thể. Trình trich xuất đi kèm với một bộ nhãn thực thể được định nghĩa trước, nhưng cũng có thể điều chỉnh chúng theo nhu cầu cụ thể của mình.

4.4 Ingestion Pipeline

Ingestion Pipeline là một công cụ giúp kết nối các bước chúng ta đã học thành một quy trình hoàn chỉnh để tạo ra các nodes từ dữ liệu ban đầu. Nó hoạt động dựa trên khái niệm "Transformations" các bước biến đổi dữ liệu. Những biến đổi này được áp dụng lên dữ liệu đầu vào để tạo ra các nodes. Sau đó, các nodes này có thể được trả về hoặc được chèn vào một cơ sở dữ liệu vector để lưu trữ.

Một điểm đặc biệt của Ingestion Pipeline là nó sử dụng cache. Điều này có nghĩa là khi bạn chạy lại quy trình với cùng một dữ liệu và biến đổi, nó sẽ sử dụng kết quả đã lưu trước đó, giúp tiết kiệm thời gian và chi phí sử dụng API.

Đoạn mã dưới đây minh họa cách sử dụng Ingestion Pipeline để xử lý một đoạn văn bản.

```

1 from llama_index.core import Document
2 from llama_index.embeddings.openai import OpenAIEmbedding
3 from llama_index.core.node_parser import SentenceSplitter
4 from llama_index.core.extractors import TitleExtractor
5 from llama_index.core.ingestion import IngestionPipeline, IngestionCache
6
7 text = """Hôm nay Sài Gòn mưa quá, thèm một chiếc kem lạnh thấu tâm can.
8      Ngoài trời mưa vẫn rơi như trút, tôi thì cạn khô!
9      """
10
11 doc = Document(text=text)
12
13 # tạo pipeline với các bước chuyển đổi
14 pipeline = IngestionPipeline(
15     transformations=[
16         SentenceSplitter(chunk_size=50, chunk_overlap=0),
17         TitleExtractor(),
18         OpenAIEmbedding(),
19     ]
20 )
21
22 # chạy pipeline
23 nodes = pipeline.run(documents=[doc])
24 for node in nodes:
25     print(node)
26     print(node.metadata)

```

```

=====
Output =====
Node ID: ca641c7a-3279-4c65-b1cc-58b48f478c1a
Text: Hôm nay Sài Gòn mưa quá, thèm một chiếc kem lạnh thấu tâm can.
{'document_title': '"Những Cơn Mưa Sài Gòn và Nỗi Nhớ Quê: Hành Trình Tìm Kiếm Ký Úc
Qua Những Giọt Mưa"'}
Node ID: b1b07eb7-99c0-44bb-959e-02370ddbb314
Text: Ngoài trời mưa vẫn rơi như trút, tôi thì cạn khô!
{'document_title': '"Những Cơn Mưa Sài Gòn và Nỗi Nhớ Quê: Hành Trình Tìm Kiếm Ký Úc
Qua Những Giọt Mưa"'}
=====
```

Giải thích các bước:

- Document: Đầu tiên, đoạn văn bản được gói gọn trong một đối tượng Document.
- SentenceSplitter: Bước đầu tiên trong pipeline là chia đoạn văn bản thành các câu nhỏ hơn (kích thước tối đa là 25 ký tự). Điều này giúp xử lý từng phần của văn bản một cách chi tiết.
- TitleExtractor: Tiếp theo, công cụ này sẽ trích xuất một tiêu đề cho từng phần văn bản. Trong ví dụ này, tiêu đề "Những Cảm Xúc Trong Mưa: Hành Trình Tìm Kiếm Vị Ngọt và Ý Nghĩa" đã được tạo ra cho tất cả các đoạn.
- OpenAIEmbedding: Cuối cùng, embeddings được tạo ra từ các đoạn văn bản. Embeddings là các biểu diễn toán học của văn bản, giúp mô hình AI hiểu và làm việc với chúng.

Chúng ta có thể lưu lại cache và load lại theo cách sau:

```

1 # Lưu trữ kết quả vào cache
2 pipeline.persist("pipeline_cache")
3 # Load kết quả từ cache
4 new_pipeline = IngestionPipeline(
5     transformations=[
6         SentenceSplitter(chunk_size=50, chunk_overlap=0),
7         TitleExtractor(),
8         OpenAIEmbedding(),
9     ]
10 )
11 new_pipeline.load("pipeline_cache")
12 nodes = new_pipeline.run(documents=[doc])
13 for node in nodes:
14     print(node)

```

```

=====
Node ID: 08b538e8-c8a9-4ff8-a1ae-5f3fec8da531
Text: Hôm nay Sài Gòn mưa quá, thèm một chiếc kem lạnh thấu tâm can.
Node ID: 671fe15f-0613-4b91-923f-bb491b081d59
Text: Ngoài trời mưa vẫn rơi như trút, tôi thì cạn khô!
=====
```

Vậy là chúng ta đã lưu và load lại pipeline thành công, điều này rất hữu ích khi triển khai ứng dụng thực tế và giảm chi phí tạo node.

4.5 Thực hành 1: Xây dựng Ingestion Pipeline - Mental Health

Tạo 1 dự án mới cấu trúc sau:

- data: thư mục lưu trữ, quản lí dữ liệu
 - cache: Thư mục lưu trữ lịch sử chat, pipeline
 - images: Thư mục chứa các hình ảnh sử dụng cho giao diện
 - index_storage: Thư mục lưu trữ quản lí index
 - ingestion_storage: Thư mục chứa dữ liệu thô
 - user_storage: Thư mục lưu trữ thông tin người dùng, kết quả đánh giá.
- src: Thư mục chứa mã nguồn chính của dự án
 - authenticate.py: Module xử lý xác thực người dùng như đăng nhập, đăng ký.
 - conversation_engine.py: Module tạo Agent, quản lí cuộc trò chuyện.

- global_settings.py: Module thiết lập các đường dẫn đến data
- index_builder.py: Module tạo index
- ingest_pipeline.py: Module tạo node
- prompts.py: Module quản lý các prompt trong dự án
- slide_bar.py: Moudule thiết lập side bar phục vụ cho việc xây dựng giao diện
- pages: Giao diện trang ứng dụng
 - 1_user.py: Giao diện kết quả
 - 2_Chat.py: Giao diện trò chuyện
- streamlit: Thư mục chứa các khóa API, tùy chỉnh cấu hình streamlit.
 - secrets.toml: File chứa OpenAI API key
- build_data.py: Tạo node, index và lưu trữ vào kho dữ liệu
- evaluate.py: Dánh giá hệ thống
- Home.py: Giao diện trang chủ của ứng dụng

Sau khi chúng ta tạo dự án với cấu trúc trên, hoặc có thể clone từ github tại [đây](#), sau đó tiến hành tạo môi trường và cài đặt các gói cần thiết, tuy nhiên bạn có thể sử dụng môi trường từ phần hướng dẫn các ví dụ trên.

Để tạo môi trường mới, chúng ta sử dụng lệnh sau:

```
1 conda create -n aio_mental_health python=3.11
```

Tiếp theo tiến hành cài đặt các thư viện:

```
1 llama_index==0.11.4
2 PyYAML
3 plotly==5.24.0
4 streamlit==1.38.0
5 docx2txt==0.8
```

Hoặc có thể cài đặt nhanh bằng cách sử dụng file requirements.txt được cung cấp kèm theo mã nguồn:

```
1 pip install -r requirements.txt
```

Sau khi thiết lập môi trường, chúng ta nhập thông tin khóa OpenAI API vào file secrets.toml với cấu trúc sau:

```
1 [openai]
2 OPENAI_API_KEY = "sk-Your-API-KEY"
```

Chúng ta tiến hành làm việc với module global_settings.py, thiết lập tên file, các đường dẫn cần thiết để phục vụ cho hoạt động lưu trữ, truy xuất dữ liệu của toàn bộ hệ thống.

```
1 CACHE_FILE = "data/cache/pipeline_cache.json"
2 CONVERSATION_FILE = "data/cache/chat_history.json"
3 STORAGE_PATH = "data/ingestion_storage/"
4 FILES_PATH = ["data/ingestion_storage/dsm-5-cac-tieu-chuan-chan-doan.docx"]
5 INDEX_STORAGE = "data/index_storage"
6 SCORES_FILE = "data/user_storage/scores.json"
7 USERS_FILE = "data/user_storage/users.yaml"
```

Trong đó:

- CACHE_FILE: Đường dẫn file cache cho pipeline, lưu trữ dữ liệu tạm thời sau khi xử lý.

- CONVERSATION_FILE: Đường dẫn file lưu lịch sử hội thoại của người dùng.
- STORAGE_PATH: Thư mục chứa dữ liệu đầu vào cần xử lý.
- FILES_PATH: Danh sách file dữ liệu đầu vào cụ thể, ví dụ tài liệu DSM-5.
- INDEX_STORAGE: Thư mục lưu trữ các index đã tạo ra.
- SCORES_FILE: Đường dẫn file lưu điểm số của người dùng.
- USERS_FILE: Đường dẫn file lưu thông tin người dùng.

Tiếp theo, chúng ta sẽ làm việc với module `ingest_pipeline.py`. Module này hỗ trợ chúng ta xây dựng một pipeline để xử lý dữ liệu thô, tạo node và lưu trữ cache của pipeline vào `CACHE_FILE` để có thể tải lại sau này.

```

1 from llama_index.core import SimpleDirectoryReader
2 from llama_index.core.ingestion import IngestionPipeline, IngestionCache
3 from llama_index.core.node_parser import TokenTextSplitter
4 from llama_index.core.extractors import SummaryExtractor
5 from llama_index.embeddings.openai import OpenAIEmbedding
6 from llama_index.core import Settings
7 from llama_index.llms.openai import OpenAI
8 import openai
9 import streamlit as st
10 from src.global_settings import STORAGE_PATH, FILES_PATH, CACHE_FILE
11 from src.prompts import CUSTOM_SUMMARY_EXTRACT_TEMPLATE
12
13 openai.api_key = st.secrets.openai.OPENAI_API_KEY
14 Settings.llm = OpenAI(model="gpt-4o-mini", temperature=0.2)
15
16 def ingest_documents():
17     # Load documents, easy but we can't move data or share for another device.
18     # Because document id is root file name when our input is a folder.
19     # documents = SimpleDirectoryReader(
20     #     STORAGE_PATH,
21     #     filename_as_id = True
22     # ).load_data()
23
24     documents = SimpleDirectoryReader(
25         input_files=FILES_PATH,
26         filename_as_id = True
27     ).load_data()
28     for doc in documents:
29         print(doc.id_)
30
31     try:
32         cached_hashes = IngestionCache.from_persist_path(
33             CACHE_FILE
34         )
35         print("Cache file found. Running using cache...")
36     except:
37         cached_hashes = ""
38         print("No cache file found. Running without cache...")
39 pipeline = IngestionPipeline(
40     transformations=[
41         TokenTextSplitter(
42             chunk_size=512,
43             chunk_overlap=20
44         ),
45         SummaryExtractor(summaries=['self'], prompt_template=
CUSTOM_SUMMARY_EXTRACT_TEMPLATE),
46         OpenAIEmbedding()
47     ],
48     cache=cached_hashes
49 )
50
51 nodes = pipeline.run(documents=documents)
52 pipeline.cache.persist(CACHE_FILE)
53
54 return nodes

```

Chương trình trên hoạt động như sau:

- Đầu tiên là lấy API KEY từ file secrets bằng streamlit và thiết lập llm, mô hình chúng ta sử dụng là gpt-4o-mini với mức độ ngẫu nhiên 0.2
- Tiếp theo hàm ingest_documents() thực hiện đọc dữ liệu thô bằng công cụ SimpleDirectoryReader, trong trường hợp này, đầu vào chúng ta sử dụng là danh sách files dữ liệu cũ thẻ, vì nếu dùng đầu vào là đường dẫn thư mục như các ví dụ trước, id của documents được tạo ra sẽ có giá trị là đường dẫn tuyệt đối đến file, chính vì vậy khi chúng ta muốn chia sẻ dữ liệu sau khi xử lý cho người khác hoặc muốn dùng lại trong dự án khác sẽ không sử dụng được.
- Sau đó thì đến phần pipeline để xử lý dữ liệu, đầu tiên chúng ta kiểm tra xem đã có dữ liệu cache pipeline nào trước đó chưa, nếu có rồi thì load lại, nếu chưa có thì tạo cache mới. Sau đó tiến hành tạo pipeline sử dụng kỹ thuật chia nhỏ dữ liệu TokenTextSplitter với chunk_size=512, chunk_overlap=20. Một số thí nghiệm trên llamaindex chỉ ra rằng giá trị chunk_size=512 hoặc 1024 sẽ cho kết quả tốt.
- Tiếp theo chúng ta sử dụng kỹ thuật khai thác metadata với SummaryExtractor, trong dự án này chúng ta chỉ tạo tóm tắt tại chính node hiện tại vì vậy chúng ta sử dụng tham số self, ngoài ra vẫn đề đã đề cập trong ví dụ trước là do prompt mặc định của SummaryExtractor viết bằng tiếng anh, trong khi chúng ta đang làm dự án với dữ liệu tiếng Việt, vì vậy chúng ta cần thiết lập lại prompt là tiếng Việt. Phần thiết lập prompt bạn mở module prompts.py và thiết lập prompt như sau:

```

1 CUSTORM_SUMMARY_EXTRACT_TEMPLATE = """\
2 Dưới đây là nội dung của phần:
3 {context_str}
4
5 Hãy tóm tắt các chủ đề và thực thể chính của phần này.
6
7 Tóm tắt: """
8

```

- Embedding: Thật ra đối với phương pháp không sử dụng pipeline thì phần này thuộc giai đoạn tạo index, tuy nhiên khi sử dụng pipeline thì bắt buộc phải tạo embedding trong này. Theo mặc định thì mô hình chúng ta sử dụng từ API là model – text-embedding-ada-002
- Cuối cùng chúng ta tạo node bằng cách chạy pipeline vừa tạo và lưu lại bộ nhớ đệm của pipeline trong CACHE_FILE.

5 Kỹ thuật Indexing Data

Chỉ mục (Index) là một cấu trúc dữ liệu quan trọng giúp chúng ta nhanh chóng truy xuất ngữ cảnh liên quan đến truy vấn của người dùng. Trong LlamaIndex, chỉ mục là nền tảng chính cho các ứng dụng tăng cường truy xuất (RAG), cho phép trả lời câu hỏi và trò chuyện với dữ liệu của bạn. Code phần này có tại: [docs/tutorial/3.Indexing.ipynb](#)

Ứng dụng của Chỉ mục:

- Tạo Query Engine và Chat Engine: Chỉ mục giúp xây dựng các công cụ truy vấn và trò chuyện, giúp người dùng dễ dàng đặt câu hỏi và nhận câu trả lời từ dữ liệu.
- Lưu trữ Dữ liệu: Chỉ mục lưu trữ dữ liệu dưới dạng các đối tượng Node, đại diện cho các phần nhỏ của tài liệu gốc. Điều này giúp truy xuất thông tin nhanh chóng và hiệu quả.

Có nhiều loại chỉ mục khác nhau như: Summary Index, Vector Store Index, Tree Index, Keyword Table Index, Property Graph Index. Trong phần này chúng ta sẽ tập trung vào Vector Store Index, lưu trữ và sử dụng nó.

5.1 Tạo Vector Store Index

Vector Store Index là một loại chỉ mục lưu trữ mỗi Node (một phần của tài liệu gốc) cùng với một embedding tương ứng trong một kho lưu trữ vector. Điều này giúp chúng ta dễ dàng truy xuất thông tin liên quan từ các đoạn văn bản bằng cách so sánh mức độ tương đồng giữa các embeddings.

Khi ta truy vấn một Vector Store Index, hệ thống sẽ tìm kiếm và lấy ra top-k các Node có mức độ tương đồng cao nhất với truy vấn của ta. Những Node này sau đó sẽ được đưa vào Response Synthesis module để tạo ra câu trả lời hoặc phản hồi thích hợp cho truy vấn.

Có hai cách tạo index, từ document và từ node, quá trình tạo index sẽ sử dụng model embedding mặc định của OPENAI là TEXT_EMBED_ADA_002. Ví dụ sau sẽ tạo index từ document:

```

1 from llama_index.core import Document
2 from llama_index.core import VectorStoreIndex
3
4 text = """Mèo rất đáng yêu.
5 Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.
6 Ở dạng đậm đặc, acid này có thể tạo thành các sương mù acid,
7 chúng đều có khả năng ăn mòn các mô con người, gây tổn thương cơ quan hô hấp,
8 mắt, da và ruột."""
9 doc = Document(text=text)
10 index = VectorStoreIndex.from_documents([doc])

```

Cách thứ 2, tạo index từ node:

```

1 from llama_index.core import Document
2 from llama_index.embeddings.openai import OpenAIEmbedding
3 from llama_index.core.node_parser import SentenceSplitter
4 from llama_index.core.extractors import TitleExtractor, QuestionsAnsweredExtractor
5 from llama_index.core.ingestion import IngestionPipeline, IngestionCache
6
7
8 text = """Mèo rất đáng yêu.
9 Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.
10 Ở dạng đậm đặc, acid này có thể tạo thành các sương mù acid,
11 chúng đều có khả năng ăn mòn các mô con người, gây tổn thương cơ quan hô hấp,
12 mắt, da và ruột."""
13
14 doc = Document(text=text)

```

```

1 # create the pipeline with transformations
2 pipeline = IngestionPipeline(
3     transformations=[
4         SentenceSplitter(chunk_size=50, chunk_overlap=0),
5         TitleExtractor(),
6         OpenAIEmbedding(),
7     ]
8 )
9
10 # run the pipeline
11 nodes = pipeline.run(documents=[doc])
12
13 # Create index from nodes
14 index = VectorStoreIndex(nodes)

```

5.2 Lưu trữ Index

a) Vector Store

```

1 from llama_index.core import StorageContext, load_index_from_storage
2
3 # Lưu trữ index vào thư mục
4 index.storage_context.persist(persist_dir="index_cache")
5
6 # Load index từ thư mục
7 storage_context = StorageContext.from_defaults(
8     persist_dir="index_cache")
9 reload_index = load_index_from_storage(storage_context)

```

Trong chương trình trên, đầu tiên chúng ta sẽ lưu lại dữ liệu index vào thư mục index_cache. Thông tin mặc định được lưu bao gồm vector index, doc store, tree index(file này sẽ trống vì chúng ta không dùng tree, nó được tạo mặc định).

- index.storage_context: Đây là một đối tượng liên quan đến ngữ cảnh lưu trữ (storage context) của chỉ mục. Nó lưu trữ thông tin về cách dữ liệu của chỉ mục được quản lý và lưu trữ.
- persist(persist_dir="index_cache"): Phương thức persist được gọi để lưu trữ toàn bộ chỉ mục vào một thư mục được chỉ định, trong trường hợp này là "index_cache". Tất cả các thông tin cần thiết cho chỉ mục, bao gồm dữ liệu vector và các siêu dữ liệu (metadata), được lưu trữ để có thể tái sử dụng.

Sau đó để tải lại index thì chúng ta thực hiện tạo storage context từ kho lưu trữ index_cache. Cuối cùng sử dụng hàm load_index_from_storage(storage_context) để tải lại index từ storage context. Hàm này sử dụng thông tin từ storage_context để xác định vị trí và cách tải lại chỉ mục đã được lưu. Kết quả là chỉ mục đã được khôi phục.

b) Vector Database

Trong phần này, chúng ta sẽ sử dụng Chroma Database để lưu trữ vector index.

```

1 import chromadb
2 from llama_index.vector_stores.chroma import ChromaVectorStore
3 from llama_index.core import VectorStoreIndex, StorageContext, Document
4
5
6 db = chromadb.PersistentClient(path="database")
7 chroma_collection = db.get_or_create_collection("my_chroma_store")
8
9 vector_store = ChromaVectorStore(
10     chroma_collection=chroma_collection
11 )
12 storage_context = StorageContext.from_defaults(
13     vector_store=vector_store
14 )
15
16 text = "I love my cat!"
17 doc = Document(text=text)
18 index = VectorStoreIndex.from_documents(
19     documents=[doc],
20     storage_context=storage_context,
21 )

```

Trong đoạn code trên:

- Đầu tiên chúng ta tạo hoặc kết nối tới cơ sở dữ liệu vector (Chroma) tại đường dẫn "database"
- Tiếp theo ta tạo hoặc lấy một collection có tên "my_chroma_store" trong cơ sở dữ liệu để lưu trữ vector.
- Ta tiếp tục tạo vector store dựa trên collection trong Chroma, nơi các vector sẽ được lưu trữ
- Sau đó tạo storage context mặc định từ vector store, quản lý việc lưu trữ và truy xuất dữ liệu vector.
- Cuối cùng, ta tạo index với đối số là danh sách doc và storage context

Để tải lại index thì chúng ta thực hiện kết nối, load vector store, storage context như bước trên. Sau đó sử dụng VectorStoreIndex.from_vector_store để tạo lại index.

```

1 reload_index = VectorStoreIndex.from_vector_store(
2     vector_store=vector_store,
3     storage_context=storage_context
4 )

```

5.3 Thực hành 2: Xây dựng Indexing - Mental Heath

Tiếp nối bài thực hành 1, ta sẽ làm việc với module index_builder.py. Xây dựng hàm build_indexes(nodes) để tải hoặc tạo mới các index từ các node đầu vào, và sau đó lưu trữ chúng trong một thư mục xác định để sử dụng lại sau này.

```

1 from llama_index.core import VectorStoreIndex, load_index_from_storage
2 from llama_index.core import StorageContext
3 from src.global_settings import INDEX_STORAGE

```

Đầu tiên chúng ta import các module cần thiết:

- VectorStoreIndex và load_index_from_storage từ LlamaIndex để tạo và tải index.
- StorageContext từ LlamaIndex để quản lý bối cảnh lưu trữ.
- INDEX_STORAGE từ module global_settings, chứa đường dẫn đến thư mục lưu trữ index.

```

1 def build_indexes(nodes):
2     try:
3         storage_context = StorageContext.from_defaults(
4             persist_dir=INDEX_STORAGE
5         )
6         vector_index = load_index_from_storage(
7             storage_context, index_id="vector"
8         )
9         print("All indices loaded from storage.")
10    except Exception as e:
11        print(f"Error occurred while loading indices: {e}")
12        storage_context = StorageContext.from_defaults()
13        vector_index = VectorStoreIndex(
14            nodes, storage_context=storage_context
15        )
16        vector_index.set_index_id("vector")
17        storage_context.persist(
18            persist_dir=INDEX_STORAGE
19        )
20        print("New indexes created and persisted.")
21    return vector_index

```

Trong khối mã try

- Tạo ngõ cảnh lưu trữ (StorageContext): Đầu tiên, hàm cố gắng tạo một storage_context từ thư mục lưu trữ mặc định (INDEX_STORAGE). Đây là bối cảnh lưu trữ, giúp quản lý và tương tác với các dữ liệu đã được lưu trữ trước đó.
- Tải lại index từ store: Hàm cố gắng tải index từ storage_context với index_id="vector". Nếu việc tải thành công, nghĩa là các index đã tồn tại và được tải từ thư mục lưu trữ, thông báo "All indices loaded from storage." sẽ được in ra.

Xử lý ngoại lệ:

- Ngoại lệ: Nếu có lỗi xảy ra trong quá trình tải index (chẳng hạn như không có index nào tồn tại trong lưu trữ), thì ngoại lệ sẽ được bắt và một thông báo lỗi sẽ được in ra.
- Tạo mới các index: Khi không thể tải các index từ kho lưu trữ, một storage_context mới sẽ được tạo. Sau đó, một VectorStoreIndex mới sẽ được tạo từ các nodes đầu vào và được liên kết với storage_context này.
- index_id: Chúng ta chỉ định là vector vì trong kho dữ liệu chứa nhiều loại index khác nhau, chẳng hạn như tree index.
- Lưu trữ index mới: Cuối cùng, các index mới sẽ được lưu trữ vào thư mục được chỉ định (INDEX_STORAGE), và thông báo "New indexes created and persisted." sẽ được in ra để xác nhận quá trình này.
- Trả về: Hàm build_indexes() sẽ trả về vector_index, index được tải hoặc tạo mới, để sử dụng trong các phần tiếp theo của chương trình.

6 Kỹ thuật truy xuất dữ liệu

Trong phần 3.4 Truy vấn dữ liệu, chúng ta đã được giới thiệu về công cụ truy vấn query engine, cách sử dụng nó thật dễ dàng và là công cụ mà chúng ta sẽ ưu tiên hàng đầu khi muốn nhanh chóng có một pipeline hoàn chỉnh cho dự án. Tuy nhiên, công cụ này có nhược điểm là khó có thể tùy chỉnh. Trong phần này, chúng ta tìm hiểu cơ bản các thành phần của công cụ truy vấn:

- retrieval: Tìm kiếm
- postprocessing: Hậu xử lý
- response synthesis: Tổng hợp và phản hồi

Chúng ta sẽ tìm hiểu qua ví dụ xây dựng công cụ truy vấn dưới đây. Code phần này có tại: [docs/tutorial/4.Retriever.ipynb](#)

Đầu tiên, bạn cần nhập các module cần thiết để thực hiện việc truy xuất, xử lý và tổng hợp câu trả lời. Ở đây có một số công cụ mà chúng ta cần chú ý:

- VectorIndexRetriever, là thành phần chịu trách nhiệm tìm kiếm và truy xuất các tài liệu dựa trên vector embedding đã tạo trước đó.
- SimilarityPostprocessor dùng để xử lý kết quả sau khi truy vấn, đảm bảo rằng các kết quả trả về có độ tương đồng cao với truy vấn ban đầu.
- RetrieverQueryEngine là công cụ chính được sử dụng để thực hiện truy vấn, kết hợp các thành phần trên để tìm kiếm, xử lý, và tổng hợp kết quả.

```

1 from llama_index.core import Document
2 from llama_index.embeddings.openai import OpenAIEmbedding
3 from llama_index.core.node_parser import SentenceSplitter
4 from llama_index.core.extractors import TitleExtractor
5 from llama_index.core.ingestion import IngestionPipeline
6 from llama_index.core.retrievers import VectorIndexRetriever
7 from llama_index.core.postprocessor import SimilarityPostprocessor
8 from llama_index.core.query_engine import RetrieverQueryEngine
9 from llama_index.core import (
10     SummaryIndex, SimpleDirectoryReader, get_response_synthesizer
11 )

```

Tiếp theo, chúng ta sẽ tạo document và index. Ở đây các bước đều giống ở phần hướng dẫn tạo Ingest Pipeline, chỉ có khác là chúng ta sử dụng TitleExtractor để trích xuất tiêu đề từ tài liệu. Tiêu đề có thể giúp cải thiện khả năng truy vấn và sắp xếp thông tin.

```

1 text = """Mèo rất đáng yêu.
2     Acid hydrochloric đậm đặc nhất có nồng độ tối đa là 40%.
3     Ở dạng đậm đặc, acid này có thể tạo thành các sương mù acid,
4     chúng đều có khả năng ăn mòn các mô con người, gây tổn thương cơ quan hô hấp,
5     mắt, da và ruột."""
6
7 doc = Document(text=text)

```

```

1 # create the pipeline with transformations
2 pipeline = IngestionPipeline(
3     transformations=[
4         SentenceSplitter(chunk_size=50, chunk_overlap=0),
5         TitleExtractor(),
6         OpenAIEmbedding(),
7     ]
8 )
9
10 # run the pipeline
11 nodes = pipeline.run(documents=[doc])
12
13 # Create index from nodes
14 index = VectorStoreIndex(nodes)

```

Trong hệ thống truy vấn, việc thiết lập các thành phần như bộ truy xuất (retriever), bộ tổng hợp phản hồi (response synthesizer), và bộ xử lý hậu kỳ (postprocessor) là bước tiếp theo quan trọng. Mỗi thành phần này đóng vai trò cụ thể trong quá trình tìm kiếm và trả về kết quả. Chúng ta sẽ tạo công cụ truy xuất với VectorIndexRetriever, đây là bộ truy xuất sử dụng chỉ mục (index) đã tạo để tìm kiếm các đoạn văn bản liên quan. Và chúng ta chỉ định rằng bộ truy xuất sẽ trả về 2 kết quả có độ tương đồng cao nhất với truy vấn. Số lượng này có thể thay đổi tùy theo nhu cầu.

```

1 retriever = VectorIndexRetriever(
2     index=index,
3     similarity_top_k=2,
4 )

```

Tiếp theo ta thiết lập Bộ Tổng hợp Phản hồi (Response Synthesizer). Ta sử dụng get_response_synthesizer là hàm được sử dụng để tạo ra một bộ tổng hợp phản hồi, chịu trách nhiệm kết hợp các kết quả tìm kiếm thành một phản hồi cuối cùng. Với tham số response_mode="tree_summarize", là tham số chỉ định phản hồi sẽ được tổng hợp dưới dạng tóm tắt theo cấu trúc cây. Tham số verbose=True được sử dụng để hiển thị thêm chi tiết quá trình thực hiện truy vấn của hệ thống, giúp theo dõi và hiểu rõ hơn về cách phản hồi được tạo ra.

```

1 response_synthesizer = get_response_synthesizer(
2     response_mode="tree_summarize",
3     verbose=True
4 )

```

Cuối cùng chúng ta thiết lập bộ Xử lý hậu kỳ (Postprocessor). Chúng ta sử dụng SimilarityPostprocessor, là bộ xử lý hậu kỳ giúp lọc và tinh chỉnh kết quả sau khi đã được tìm kiếm và tổng hợp. Với tham số similarity_cutoff=0.5 là giá trị ngưỡng (cutoff) giúp xác định mức độ tương đồng tối thiểu mà một kết quả phải đạt được. Nếu độ tương đồng của một kết quả thấp hơn 0.5, kết quả đó sẽ bị loại bỏ khỏi danh sách phản hồi cuối cùng.

```

1 pp = SimilarityPostprocessor(similarity_cutoff=0.5)

```

Kết hợp các thành phần đã tạo, chúng ta có được công cụ truy vấn query_engine.

```

1 query_engine = RetrieverQueryEngine(
2     retriever=retriever,
3     response_synthesizer=response_synthesizer,
4     node_postprocessors=[pp]
5 )

```

Bây giờ, chúng ta có thể thực hiện truy vấn bằng công cụ vừa xây dựng và xem kết quả.

```

1 response = query_engine.query(
2     "Mèo rất đáng yêu phải không?"
3 )
4 print(response)

```

```
1 ====== Output ======
2 1 text chunks after repacking
3 Đúng vậy, mèo rất đáng yêu.
4 ======
```

Trên đây là hướng dẫn về cách xây dựng và tinh chỉnh một hệ thống truy vấn dữ liệu. Đối với các dự án thông thường, chỉ cần query_engine mặc định là đủ, tuy nhiên khi cần tinh chỉnh, chúng ta cần bóc tách nó ra như hướng dẫn trên để thử nghiệm các kỹ thuật khác.

7 Kỹ thuật xây dựng Chatbot, Agent

Trước khi bắt đầu tìm hiểu các kỹ thuật xây dựng Agent, chúng ta cùng tổng hợp lại một số điểm nổi bật về công nghệ chatbot cho đến các mô hình ngôn ngữ lớn hiện tại.

Năm 1964-1967 tại MIT, chatbot đầu tiên ra đời bởi Joseph Weizenbaum, có tên là Eliza. Và điều đặc biệt là chatbot này xây dựng với mục đích tham vấn trị liệu tâm lý. Tuy Chatbot này hoạt động dựa trên việc so khớp theo các chuỗi, nhưng những người dùng sử dụng nó tại thời điểm đó còn nhầm tưởng nó là chuyên gia tâm lý thật sự.

Những năm gần đây, công nghệ chatbot ở Việt Nam được ứng dụng rất rộng rãi trong các lĩnh vực kinh doanh, chăm sóc khách hàng... Tuy nhiên chúng không thông minh, thường trả lời một cách vụng về hoặc trả lời theo kịch bản được thiết lập trước khiến cuộc trò chuyện không tự nhiên.

Với sự phát triển bùng nổ của công nghệ trí tuệ nhân tạo AI, các mô hình ngôn ngữ lớn ra đời có khả năng hiểu ngôn ngữ của con người và rất thông minh. Đây là giải pháp hiệu quả để giải quyết các vấn đề mà các công nghệ chatbot cũ không đáp ứng được.

Tuy nhiên các mô hình ngôn ngữ lớn hiện nay vẫn tồn tại các vấn đề riêng của nó. Ví dụ như việc trả lời không chính xác và hiện tượng ảo giác khi mà mô hình trả lời về một vấn đề nó chưa từng được học. Tất nhiên là cũng có các giải pháp để giải quyết vấn đề này như đào tạo lại hoặc sử dụng kỹ thuật RAG để cung cấp thêm dữ liệu và tinh chỉnh dựa trên prompt - và RAG cũng chính là kỹ thuật mà chúng ta tìm hiểu suốt từ đầu bài viết đến giờ.

Sau cùng mọi công nghệ chatbot có phức tạp đến đâu chăng nữa thì cũng quy về bản nguyên chính của chatbot là hội thoại - trò chuyện với con người một cách thông minh. Trong các phần trước, sau khi xây dựng index, chúng ta có thể tạo cuộc trò chuyện bằng cách sử dụng query_engine, một công cụ truy vấn dựa trên dữ liệu của chúng ta. Tuy nhiên công cụ này không có cơ chế nào để lưu trữ lịch sử trò chuyện, trong khi trong một cuộc trò chuyện muốn duy trì thì ngữ cảnh từ lịch sử trò chuyện rất quan trọng. Code phần này có tại: [docs/tutorial/5.ChatbotAgent.ipynb](#)

7.1 Chat engine

Trong LlamaIndex, có một công cụ hiệu quả hơn để thực hiện một cuộc trò chuyện thực tế là ChatEngine, nó sử dụng dữ liệu của chúng ta và cả lịch sử trò chuyện để đưa ra phản hồi.

Cú pháp:

```
1 chat_engine = index.as_chat_engine(chat_mode, llm)
```

Cú pháp trên thực chất là để chuyển đổi chỉ mục (index) thành một công cụ trò chuyện.

Cụ thể, nó hoạt động như sau:

- Tạo công cụ truy vấn (query engine): Đầu tiên, chỉ mục (index) được sử dụng để tạo ra công cụ truy vấn, cho phép tra cứu thông tin.
- Đóng gói: Sau đó, công cụ truy vấn này được bọc trong một công cụ trò chuyện, dựa trên chế độ trò chuyện (chat_mode). Điều này cho phép ta tương tác với hệ thống như thể đang trò chuyện với một người thật, trong đó các câu trả lời sẽ dựa trên thông tin từ chỉ mục.

Dưới đây là giải thích tham số:

- chat_mode: Là chế độ trò chuyện, có nhiều loại để chúng ta lựa chọn:
 - ChatMode.BEST (mặc định): Công cụ trò chuyện sử dụng tác nhân (react hoặc openai) với công cụ công cụ truy vấn
 - ChatMode.CONTEXT: Công cụ trò chuyện sử dụng ngữ cảnh từ index storage
 - ChatMode.CONDENSE_QUESTION: Công cụ trò chuyện cô đọng các câu hỏi

- ChatMode.CONDENSE_PLUS_CONTEXT: Công cụ trò chuyện cô đọng các câu hỏi và sử dụng trình thu thập để lấy ngữ cảnh
- ChatMode.SIMPLE: Công cụ trò chuyện đơn giản sử dụng LLM trực tiếp
- ChatMode.REACT: Công cụ trò chuyện sử dụng react agent với công cụ công cụ truy vấn
- ChatMode.OPENAI: Công cụ trò chuyện sử dụng openai agent với công cụ công cụ truy vấn
- llm: mô hình llm mà chúng ta sẽ sử dụng.

Có nhiều cách sử dụng chat engine:

- chat(): Phương thức này dùng để bắt đầu một phiên trò chuyện và sẽ trả về kết quả ngay lập tức.
- achat(): Giống như chat() nhưng tích hợp truy vấn bất đồng bộ.
- stream_chat(): Tương tự chat() nhưng kết quả trả về sẽ hiển thị thông output trực tiếp từ API. Tức là nó giống giao diện chatGPT, kết quả không hiển thị toàn bộ mà nó sẽ xuất hiện từ từ(các chữ như đang chạy ra từ màn hình). Điều này có thể tăng trải nghiệm tương tác với người dùng.
- astream_chat(): Tương tự streaming_chat() nhưng sử dụng cơ chế bất đồng bộ.
- chat_repl(): Tạo một cuộc trò chuyện liên tục trả lời câu hỏi của người dùng cho đến khi người dùng kết thúc cuộc trò chuyện.

Dưới đây là ví dụ về cách sử dụng chat(), astream_chat(), và chat_repl(). Đầu tiên chúng ta sẽ tạo chat_engine từ index đơn giản sau:

```

1 from llama_index.core import Document, VectorStoreIndex
2
3 text = "Tôi có một bé mèo, cậu ấy tên là Mèo Ú. Cậu ấy đang ngủ bên cửa sổ."
4     Tôi rất quý cậu ấy."
5 doc = Document(text=text)
6 index = VectorStoreIndex.from_documents([doc])
7 chat_engine = index.as_chat_engine()

```

Tiếp theo, chúng ta sẽ sử dụng phương thức chat() để bắt đầu cuộc trò chuyện.

```

1 response = chat_engine.chat("Tôi là Tiềm,          ===== Output =====
                                mèo Ú của tôi đang làm gì nhỉ?")
2 print(response)               Mèo Ú của bạn đang ngủ bên cửa sổ.
                                =====

```

Như đã giới thiệu, chat_engine sử dụng dữ liệu truy vấn và cả lịch sử trò chuyện trong cuộc hội thoại. Điều đó thể hiện rõ qua ví dụ sau:

```

1 response = chat_engine.chat("Tôi tên là gì      ===== Output =====
                                ?")                     Bạn tên là Tiềm.
2 print(response)               =====

```

Bạn có thể thấy rằng, tên mình không có trong dữ liệu ban đầu, nhưng khi trò chuyện mình cố tình nói tên mình trong lần trò chuyện trước. Vậy nên khi tiếp tục trò chuyện, mình đã hỏi tên mình là gì, và chat engine đã trả lời đúng, điều này là do chat engine đã sử dụng nội dung lịch sử trò chuyện trước đó. Chúng ta có thể xem chi tiết lịch sử cuộc trò chuyện trên bằng cách sử dụng hàm chat_history như sau:

```
1 chat_engine.chat_history
```

```
===== Output =====
[ChatMessage(role=<MessageRole.USER: 'user'>, content='Tôi là Tiềm, mèo Ú của tôi đang
làm gì nhỉ?', additional_kwargs={}),
 ChatMessage(role=<MessageRole.ASSISTANT: 'assistant'>, content=None,
 additional_kwargs={'tool_calls': [ChatCompletionMessageToolCall(id='
call_SztjRMkbEtABahzyhVb5CSGU', function=Function(arguments='{"input": "Mèo Ú của
Tiềm đang làm gì?"}', name='query_engine_tool'), type='function')])),
 ChatMessage(role=<MessageRole.TOOL: 'tool'>, content='Mèo Ú đang ngủ bên cửa sổ.',
 additional_kwargs={'name': 'query_engine_tool', 'tool_call_id': ''
call_SztjRMkbEtABahzyhVb5CSGU'}),
 ChatMessage(role=<MessageRole.ASSISTANT: 'assistant'>, content='Mèo Ú của bạn
đang ngủ bên cửa sổ.', additional_kwargs={}),
 ChatMessage(role=<MessageRole.USER: 'user'>, content='Tôi tên là gì?',
 additional_kwargs={}),
 ChatMessage(role=<MessageRole.ASSISTANT: 'assistant'>, content='Bạn tên là Tiềm.',
 additional_kwargs={})
=====
```

Tương tự như chat(), dưới đây là ví dụ về cách sử dụng stream_chat().

```
1 response = chat_engine.stream_chat("Tôi có yêu cầu ấy không? Viết một bài thơ về Mèo Ú
  đi.")
2 for token in response.response_gen:
3     print(token, end="")
```

```
===== Output =====
Có vẻ như bạn đang muốn biết về tình cảm của mình đối với Mèo Ú. Dưới đây là một bài
thơ thể hiện tình yêu dành cho Mèo Ú:

Mèo Ú nhỏ bé, đáng yêu vô cùng,
Trong lòng tôi, cậu là ánh trăng rằm.
Mỗi buổi sáng, cậu nhảy nhót vui,
Mang đến nụ cười, xua tan u ám.

Lông mềm mại, như mây trời bay,
Ánh mắt trong veo, như nước hồ đầy.
Cậu là bạn thân, là niềm an ủi,
Mỗi phút giây bên nhau, thật tuyệt vời.

Khi cậu ngủ say, tôi ngắm nhìn,
Tình yêu dành cho cậu, mãi không phai.
Mèo Ú ơi, cậu là món quà,
Trong trái tim tôi, cậu luôn là nhất.
=====
```

Cuối cùng, hãy đến ví dụ về cách sử dụng chat_repl(), ngoài việc khác biệt so với chat() hay stream_chat() bởi giao diện hoạt động như một vòng lặp, nó còn khác biệt bởi cơ chế lưu trữ lịch sử chat, nó khởi tạo mới lịch sử chat khi bắt đầu một phiên chat mới. Chính vì vậy phiên chat mới không sử dụng lại lịch sử của phiên chat cũ.

```
1 response = chat_engine.chat_repl()
```

```
===== Output =====
===== Entering Chat REPL =====
Type "exit" to exit.

User: Chào bạn
Assistant: Chào bạn! Bạn cần giúp gì hôm nay?

User: Bạn có biết mèo ú không?
Assistant: Mèo ú thường được biết đến với vẻ ngoài dễ thương và tính cách thân thiện.
Bạn có kỷ niệm nào đặc biệt với mèo ú không? Hoặc bạn muốn biết thêm thông tin gì
về chúng?

User: Cậu ấy đang làm gì?
Assistant: Mèo ú đang ngủ bên cửa sổ. Có vẻ như cậu ấy đang tận hưởng một giấc ngủ
ngon! Bạn có muốn biết thêm điều gì về mèo ú không?

User: exit
=====
```

Để xóa lịch sử trò chuyện hoặc làm mới lại chat_engine, chúng ta sử dụng chat_engine.reset().

```
1 # Xóa lịch sử chat
2 chat_engine.reset()
3 chat_engine.chat_history
```

```
===== Output =====
[]
=====
```

7.2 Hoạt động của bộ nhớ trò chuyện

ChatMemoryBuffer là một bộ nhớ đệm giúp lưu trữ và quản lý lịch sử trò chuyện hiệu quả, ngoài ra nó còn có thể được sử dụng từ phiên trò chuyện này đến phiên trò chuyện khác, đảm bảo tính bền vững của cuộc trò chuyện. Hơn nữa nó còn quản lý ngữ cảnh trò chuyện rất tốt, khi cuộc trò chuyện dài vượt quá ngữ cảnh cho phép, nó sẽ tự động cắt bớt những phần lịch sử trò chuyện cũ nhất bằng phương pháp trượt window.

Có hai cách để lưu trữ lịch sử trò chuyện là:

- SimpleChatStore: Lưu trữ lịch sử trò chuyện vào bộ nhớ Cache.
- RedisChatStore: Lưu trữ lịch sử trò chuyện trong Redis database, phương pháp này rất linh hoạt, chúng ta không cần quản lý lịch sử theo cách thủ công.

Trong ví dụ dưới đây, chúng ta sẽ tạo một cuộc trò chuyện và lưu trữ, tái sử dụng lịch sử trò chuyện với SimpleChatStore.

Chúng ta bắt đầu với việc khai báo các thư viện cần thiết, ở đây chúng ta sẽ sử dụng SimpleChatStore, ChatMemoryBuffer để lưu trữ, quản lý lịch sử cuộc hội thoại. Sau đó, chúng ta xây dựng index đơn giản từ một đoạn văn bản.

```
1 from llama_index.core.storage.chat_store import SimpleChatStore
2 from llama_index.core.chat_engine import SimpleChatEngine
3 from llama_index.core.memory import ChatMemoryBuffer
4 from llama_index.core import Document, VectorStoreIndex
5
6 text = "Mèo Ú đi lạc, mấy ngày rồi cậu ấy không về nhà, tôi lo quá!"
7 doc = Document(text=text)
8 index = VectorStoreIndex.from_documents([doc])
```

Tiếp theo, chúng ta tạo chat_store bằng cách tải lại file chứa lịch sử trò chuyện trước đó. Nếu đây là lần đầu tiên trò chuyện, chat_store sẽ được khởi tạo mới bởi SimpleChatStore, và chat_store lúc này sẽ trống rỗng.

```

1 try:
2     chat_store = SimpleChatStore.from_persist_path(
3         persist_path = "chat_memory.json"
4     )
5 except FileNotFoundError:
6     chat_store = SimpleChatStore()

```

Tiếp theo, chúng ta sẽ thiết lập bộ nhớ trong cuộc trò chuyện, theo mặc định thì nó đã được thiết lập sẵn trong chế độ chat. Nhưng ở đây, chúng ta sẽ thử tinh chỉnh bằng cách thiết lập một số tham số theo ý của mình. Khi tinh chỉnh ChatMemoryBuffer, có 3 tham số mà chúng ta cần chú ý:

- token_limit: Số lượng token tối đa cho phép trong lịch sử trò chuyện, khi vượt quá số lượng, cơ chế trượt window sẽ cắt bỏ những phần lịch sử cũ hơn.
- chat_store: Lịch sử chat trước đó
- chat_store_key: Khóa định danh xác định cuộc trò chuyện này với cuộc trò chuyện khác. Nó hữu ích trong trường hợp ứng dụng cung cấp cho nhiều người dùng, cuộc trò chuyện của mỗi người sẽ được gán một khóa riêng để phân biệt với nhau.

```

1 memory = ChatMemoryBuffer.from_defaults(
2     token_limit= 20,
3     chat_store=chat_store,
4     chat_store_key="User 1"
5 )

```

Cuối cùng, chúng ta tạo chat engine với tham số memory và bắt đầu cuộc trò chuyện.

```

1 chat_engine = index.as_chat_engine(memory=memory)
2 while True:
3     user_input = input("User: ")
4     if user_input == "exit":
5         break
6     response = chat_engine.chat(user_input)
7     print("Bot:", response)

```

Chúng ta sử dụng chat_history để kiểm tra thông tin lịch sử của cuộc trò chuyện.

```
1 chat_engine.chat_history
```

Cuối cùng, để lưu lại lịch sử chat, chúng ta sử dụng persist để lưu thông tin đến file "chat_memory.json"

```
1 chat_store.persist(persist_path="chat_memory.json")
```

7.3 Các chế độ trò chuyện

a) Simple Chat Engine

Đây là một chế độ trò chuyện đơn giản, hoạt động dựa trên tương tác giữa người dùng và LLM mà không sử dụng thêm dữ liệu bên ngoài. Chúng ta có thể tinh chỉnh thông qua các tham số prompt, memory. Dưới đây, chúng ta sẽ tạo một công cụ chat đơn giản với các tham số mặc định.

```

1 from llama_index.core.chat_engine import SimpleChatEngine
2
3 chat_engine = SimpleChatEngine.from_defaults()
4 chat_engine.chat_repl()

```

b) Context Chat Engine

Chế độ trò chuyện này nâng cao hơn chế độ Simple chat engine bởi nó sử dụng kiến thức từ dữ liệu của chúng ta bằng cách truy xuất dựa trên chỉ mục. Câu trả lời được kết hợp thông tin từ lịch sử trò chuyện và nội dung truy vấn, nên câu trả lời sẽ chính xác và tự nhiên hơn.

```

1 from llama_index.core import Document, VectorStoreIndex
2
3 text = "Đêm qua đi chơi về muộn, mèo Ú bị sập bẫy!"
4 doc = Document(text=text)
5 index = VectorStoreIndex.from_documents([doc])

1 chat_engine = index.as_chat_engine(
2     chat_mode= "context",
3     system_prompt=( "Bạn là một nhà thám tử, "
4                     "hãy giúp tôi tìm ra ai đã bắt mèo Ú!")
5 )
6 chat_engine.chat_repl()

```

c) Condense Question Chat Engine

Trong cuộc trò chuyện, nhiều khi người dùng nhập câu hỏi không rõ ràng, việc này dẫn đến câu trả lời không chính xác, giống kiểu "ông nói gà bà nói vịt". Để khắc phục vấn đề này, người ta cho rằng trong cuộc hội thoại, câu hỏi hiện tại sẽ liên quan đến các thông tin trò chuyện trước đó, vì vậy họ kết hợp câu hỏi đầu vào của người dùng với lịch sử trò chuyện, thông qua mô hình ngôn ngữ lớn tạo ra một câu hỏi mới rõ ràng hơn. Sau đó thì thực hiện tìm kiếm trong kho dữ liệu của chúng ta và tạo câu trả lời từ kết quả tìm kiếm.

```

1 chat_engine = index.as_chat_engine(
2     chat_mode= "condense_question"
3 )
4 chat_engine.chat_repl()

```

Chế độ trò chuyện này có ưu điểm là mọi câu trả lời đều được tạo từ thông tin trong kho dữ liệu của chúng ta. Tuy nhiên, đây cũng chính là nhược điểm, ví dụ khi chúng ta chỉ nói "xin chào", nó cũng thực hiện các bước phức tạp nên có thể gây lãng phí tài nguyên, và cơ chế tạo lại câu hỏi có thể tạo ra câu hỏi chung chung thì cũng khó mà tìm được câu trả lời.

d) Condense Plus Context Chat Engine

Chế độ trò chuyện này là sự kết hợp giữa hai chế độ trò chuyện trên. Chế độ này sử dụng đầu ra của chế độ Condense với dữ liệu tìm kiếm được trong kho dữ liệu và đưa qua LLM một lần nữa để tạo ra câu trả lời cuối cùng. Cách này có thể đưa ra câu trả lời tốt hơn, nhưng việc sử dụng LLM hai lần trong chế độ này sẽ làm tăng thời gian phản hồi và chi phí sử dụng API.

```

1 chat_engine = index.as_chat_engine(
2     chat_mode= "condense_plus_context"
3 )
4 chat_engine.chat_repl()

```

7.4 Agent

Nếu chúng ta hỏi chatbot "Mấy giờ rồi nhỉ?", hay ta muốn chatbot viết một bức thư rồi gửi nó qua địa chỉ email của khách hàng, hoặc ta có một cơ sở dữ liệu muốn chatbot có thể kết nối và truy xuất thông tin dựa trên yêu cầu, sau đó viết thành file báo cáo rồi gửi cho ta, giống như một nhân viên thật sự?... nói chung là chúng ta muốn chatbot có thể thực hiện nhiều công việc hơn thì làm như thế nào?

Các phương pháp trước đây, khi kết hợp LLM với dữ liệu (RAG cơ bản), chỉ cho phép chatbot thực hiện chức năng trò chuyện. Điều này hạn chế khả năng của chatbot, khiến nó chỉ có thể trả lời câu hỏi hoặc cung cấp thông tin dựa trên dữ liệu tĩnh mà nó được cung cấp. Chúng ta không thể yêu cầu chatbot tìm kiếm thông tin mới, thay đổi dữ liệu hiện có, hoặc thực hiện các tác vụ phức tạp khác.

Đây là lúc mà Agent xuất hiện—một thành phần có thể thay đổi cách chúng ta nghĩ về khả năng của chatbot. Với Agent trong LlamaIndex, chatbot không chỉ có khả năng trò chuyện mà còn có thể tự động tìm kiếm thông tin, kết nối với các dịch vụ bên ngoài, và thậm chí cập nhật hoặc thay đổi dữ liệu một cách thông minh.

Agents như những người trợ lý ảo có thể thực hiện cả chức năng "đọc" và "ghi," nghĩa là chúng có thể tìm kiếm và truy xuất thông tin từ nhiều loại dữ liệu khác nhau, cũng như thay đổi hoặc cập nhật dữ liệu khi cần thiết. Điều này giúp mở rộng đáng kể những gì mà chatbot có thể làm, từ việc chỉ trả lời câu hỏi trở thành một công cụ thông minh có khả năng thực hiện nhiều tác vụ khác nhau.

Để tạo một Agent dữ liệu, ta cần các thành phần chính sau:

- Vòng lặp suy luận: Đây là cách mà agent suy nghĩ và quyết định bước tiếp theo sẽ làm gì.
- Các công cụ (Tool Abstractions): Đây là các công cụ hoặc API mà agent có thể sử dụng để lấy hoặc thay đổi thông tin. Khi ta giao một nhiệm vụ cho agent, nó sẽ sử dụng vòng lặp suy luận để quyết định sử dụng công cụ nào, theo thứ tự nào, và cách sử dụng từng công cụ.

a) Tools

Công cụ (Tools) là phần quan trọng trong việc xây dựng agent. Chúng giống như các API nhưng dành cho agent sử dụng thay vì con người. Ví dụ một tool có thể chứa query engine để truy xuất dữ liệu, hoặc chứa các hàm python để thực hiện các chức năng cụ thể, ví dụ như một hàm đọc, ghi file.

Việc chọn đúng công cụ và cách sử dụng chúng rất quan trọng vì ảnh hưởng đến cách LLM tương tác với công cụ đó. Chúng ta cần định nghĩa tên và phần mô tả khi tạo công cụ vì sẽ ảnh hưởng lớn đến việc LLM gọi và sử dụng các công cụ đó. Điều này cũng dễ hiểu vì Agent cũng giống như con người, khi đưa cho nó công cụ thì phải kèm hướng dẫn sử dụng như mô tả công cụ này là gì, chức năng như thế nào. Đối với dân lập trình Python như chúng ta không còn xa lạ gì khi mà chúng ta có những quy định đặt tên biến, hàm, viết doc-mô tả cho hàm để đọc code dễ hiểu hơn. Thông thường khi lập trình chúng ta có thể không cần mô tả, nhưng các hàm cho agent sử dụng bắt buộc phải viết để Agent có thể hiểu và sử dụng công cụ chính xác.

Có một số loại tool sau:

- FunctionTool: Chuyển đổi bất kỳ hàm nào của người dùng thành một Tool.
- QueryEngineTool: Gói gọn một công cụ truy vấn hiện có thành Tool.
- ToolSpecs từ cộng đồng: Công cụ do cộng đồng đóng góp, bao gồm nhiều dịch vụ như Gmail.
- Utility Tools: Công cụ tiện ích, hỗ trợ quản lý và xử lý lượng dữ liệu lớn từ các công cụ khác.

Đầu tiên, chúng ta hãy cùng xem một ví dụ về công cụ trò chuyện cơ bản, chỉ sử dụng LLM.

```

1 from llama_index.core.chat_engine import
2     SimpleChatEngine
3
4 chat_engine = SimpleChatEngine.from_defaults()
5
6 response = chat_engine.chat("Emơi mấy giờ
7     rồi?")
8
9 print(response)

```

===== Output =====
Xin lỗi, nhưng tôi không thể kiểm tra thời gian hiện tại. Bạn có thể xem đồng hồ hoặc thiết bị của mình để biết giờ nhé!

Mô hình trò chuyện trên không thể trả lời về thời gian hiện tại, vì nó không được học và cũng không biết bây giờ là mấy giờ để trả lời. Vậy thì, hãy xây dựng một công cụ để Agent có thể xem giờ.

```

1 import datetime
2
3 def get_date_time() -> str:
4     """Get current date and time
5
6     Returns:
7         str: Current date and time
8     """
9
10    now = datetime.datetime.now()
11
12    return now.strftime("%Y-%m-%d %H:%M:%S")

```

Chúng ta sử dụng thư viện datetime để lấy thời gian hiện tại. Bằng việc tạo một hàm get_date_time() hàm này không có tham số và được gọi ý là sẽ trả về str. Bên trong hàm chúng ta cần viết doc cho nó, mô tả chức năng của hàm, các tham số nếu có và giá trị trả về. Như đã đề cập, việc ghi chú thích như vậy rất quan trọng, để Agent có thể lựa chọn chính xác công cụ.

Tiếp theo chúng ta biến hàm này thành tool, bằng cách sử dụng FunctionTool.

```

1 from llama_index.core.tools import FunctionTool
2
3 get_date_time_tool = FunctionTool.from_defaults(fn=get_date_time)

```

Cuối cùng chúng ta tạo agent và thử hỏi giờ xem agent đã trả lời thời gian được chưa.

```

1 from llama_index.agent.openai import
2     OpenAI
3
4 agent = OpenAI.from_tools(
5     tools=[get_date_time_tool],
6 )
7
8 response = agent.chat("Emơi mấy giờ rồi?")

```

===== Output =====
Bây giờ là 22 giờ 32 phút.

Vậy là bây giờ Agent đã có trả lời thời gian chính xác, điều đơn giản mà LLM không làm được.

Tiếp theo, chúng ta sẽ tìm hiểu query engine tool. Trong Agent chúng ta có thể sử dụng nhiều công cụ truy xuất dữ liệu khác nhau. Chính vì vậy mà chúng ta sẽ gọi các công cụ truy xuất này lại thành công cụ cho Agent sử dụng.

```

1 from llama_index.core.tools import QueryEngineTool
2 from llama_index.core import Document, VectorStoreIndex
3 from llama_index.agent.openai import OpenAI
4
5
6 text = "Đêm qua đi chơi về muộn, mèo Ú bị sập bẫy!"
7 doc = Document(text=text)
8 index = VectorStoreIndex.from_documents([doc])
9
10 query_engine = index.as_query_engine()
11 query_tool = QueryEngineTool.from_defaults(query_engine=query_engine,
12                                              description="Công cụ tìm kiếm thông tin về
13                                              mèo Ú")
14 agent = OpenAI.from_tools(tools=[query_tool, get_date_time_tool])
15
16 response_1 = agent.chat("Em đi mấy giờ rồi?")
17 response_2 = agent.chat("Em có biết mèo Ú của anh ở đâu không?")
18
19 print(response_1)
20 print(response_2)

```

```

=====
Output =====
Bây giờ là 23 giờ 56 phút.
Mèo Ú của anh đã bị sập bẫy khi đi chơi về muộn.
=====
```

Trong chương trình trên, chúng ta bắt đầu bằng việc khai báo các gói cần thiết, ở đây chúng ta sẽ sử dụng QueryEngineTool để gói lại các công cụ loại query engine. Tiếp theo chúng ta tạo query engine, sau đó tạo query engine tool. Cuối cùng chúng ta sẽ tạo agent từ các công cụ. Đầu ra cho thấy agent đã sử dụng công cụ khác nhau để trả lời cho mỗi câu hỏi.

b) Resoning Loop

Vòng lặp lý luận(Resoning Loop) là một thành phần quan trọng của Agent, nó là khả năng tư duy của Agent trong việc sử dụng các công cụ để thực hiện công việc được giao. Trong phần trước, chúng ta đã tạo tool và agent, khi chúng ta trò chuyện, agent có thể chọn công cụ để trả lời câu hỏi của chúng ta một cách tự động.

Vòng lặp lý luận khi nhận được câu hỏi sẽ thực hiện đánh giá bối cảnh, lựa chọn công cụ phù hợp để sử dụng. Đối với trường hợp công việc cần phải sử dụng nhiều công cụ, Agent cũng xác định trình tự sử dụng các công cụ một cách hợp lý.

c) OpenAI Agent

OpenAI Agent được thiết kế để tương tác và thực hiện các tác vụ cụ thể thông qua giao tiếp. Nó được xây dựng dựa trên API của OpenAI và có khả năng sử dụng các công cụ (tools) để thực hiện các phép tính hoặc xử lý dữ liệu theo yêu cầu. Khả năng này đã được tích hợp sẵn trong API nên sử dụng khá đơn giản.

Việc xử lý logic để chọn và sử dụng công cụ phù hợp nằm ở phía API, khi người dùng yêu cầu thực hiện nhiệm vụ nào đó, API sẽ thực hiện phân tích ngữ cảnh yêu cầu, cùng với lịch sử trò chuyện để quyết định nên sử dụng công cụ hay đưa ra câu trả lời cuối cùng.

Trong ví dụ dưới đây, chúng ta tạo Agent với thiết lập là một người bạn thân của mình. Câu ấy có thể thực hiện phép nhân hai số rất chính xác và có thể trả lời thông tin về mèo Ú.

```

1 from typing import Optional
2 from llama_index.agent.openai import OpenAIAGent
3 from llama_index.core.tools import FunctionTool
4 from llama_index.core import Document, VectorStoreIndex
5 from llama_index.core.tools import QueryEngineTool
6
7 def multiply(a: int, b: int) -> int:
8     """Trả về kết quả phép nhân a với b"""
9     return a * b
10
11 def save_result(result: int) -> str:
12     """Lưu kết quả vào file result.txt"""
13     with open("result.txt", "w") as f:
14         f.write(str(result))
15     return "Result saved!"
16
17 text = "Sáng ngày 4 tháng 9 năm 2024, mèo Ú trở về dưới cơn mưa tầm tã."
18 doc = Document(text=text)
19 index = VectorStoreIndex.from_documents([doc])
20 query_engine = index.as_query_engine()
21
22 query_tool = QueryEngineTool.from_defaults(query_engine=query_engine,
23                                              description="Công cụ tìm kiếm thông tin về"
24                                              mèo Ú")
25 multiply_tool = FunctionTool.from_defaults(fn=multiply)
26 save_result_tool = FunctionTool.from_defaults(fn=save_result)
27
28 openai_agent = OpenAIAGent.from_tools(
29     tools=[query_tool, multiply_tool, save_result_tool],
30     system_prompt="Bạn là Thu, người bạn tri kỷ, hiểu chuyện và thành thật của tôi.",
31     verbose=True
32 )
33 response = openai_agent.chat("Thu ơi, hãy nhân 5 với 3 giúp tớ nhé. Mà cậu biết mèo Ú"
34                               của tớ đã thế nào không?")
35 print(response)

```

```

=====
Added user message to memory: Thu ơi, hãy nhân 5 với 3 giúp tớ nhé. Mà cậu biết mèo Ú
của tớ đã thế nào không?
== Calling Function ==
Calling function: multiply with args: {"a": 5, "b": 3}
Got output: 15
=====

== Calling Function ==
Calling function: query_engine_tool with args: {"input": "mèo Ú"}
Got output: Mèo Ú là một con mèo đã trở về vào sáng ngày 4 tháng 9 năm 2024, dưới cơn
mưa tầm tã.
=====

Kết quả của phép nhân 5 với 3 là 15.

Còn về mèo Ú của cậu, nó đã trở về vào sáng ngày 4 tháng 9 năm 2024, dưới cơn mưa tầm
tã. Hy vọng mèo Ú của cậu đã an toàn và khỏe mạnh!
=====
```

Trong chương trình trên:

- Đầu tiên chúng ta import các thư viện cần thiết: Đoạn code sử dụng một số thư viện từ LlamaIndex để tạo ra một agent có thể thực hiện các tác vụ cụ thể.
- Định nghĩa hàm multiply: Hàm này nhận hai số nguyên và trả về kết quả phép nhân của chúng.
- Định nghĩa hàm save_result: Hàm này nhận một số nguyên, lưu kết quả vào tệp result.txt, và trả về thông báo xác nhận đã lưu kết quả.
- Tạo tài liệu và index: Đoạn văn bản về "mèo Ú" được chuyển thành một đối tượng Document, sau đó được đưa vào VectorStoreIndex để có thể thực hiện tìm kiếm thông tin.
- Tạo công cụ QueryEngineTool: Công cụ này cho phép agent tìm kiếm thông tin liên quan đến "mèo Ú" trong documents đã được index.
- Tạo công cụ FunctionTool: Hai công cụ khác là multiply_tool để thực hiện phép nhân và save_result_tool để lưu kết quả vào tệp.
- Tạo OpenAI Agent: Agent này được cấu hình với các công cụ đã tạo, cùng với lời nhắc hệ thống rằng nó là "Thu, người bạn tri kỷ". Agent có khả năng thực hiện các tác vụ như nhân số, lưu kết quả và tìm kiếm thông tin khi được yêu cầu.
- Cuối cùng thực hiện trò chuyện với agent: Agent nhận yêu cầu từ người dùng (nhân 5 với 3 và tìm thông tin về mèo Ú) và trả về kết quả tương ứng.

d) React Agent

Khác với OpenAI Agent được tích hợp vòng lặp lý luận vào model, React Agent hoạt động dựa trên prompt, việc chọn lựa và tạo quy trình làm việc dựa trên hướng dẫn của prompt, chính vì vậy có thể áp dụng phương pháp này cho các loại llms khác nhau. Sự phức tạp đã có người khác giải quyết, chúng ta hãy xem cách sử dụng React Agent như thế nào qua ví dụ sau:

```

1 from llama_index.core.agent.react import ReActAgent
2
3 react_agent = ReActAgent.from_tools(
4     tools=[query_tool, multiply_tool, save_result_tool],
5     system_prompt="Bạn là Thu, người bạn tri kỷ, hiểu chuyện và thành thật của tôi.",
6     verbose=True
7 )
8 response = react_agent.chat("Thu ơi, hãy nhân 5 với 3 giúp tớ nhé. Mà cậu biết mèo Ú
9 của tớ đã thế nào không?")
10 print(response)

```

```
=====
> Running step d2be793c-827f-45c3-8bd8-4799bbd24722. Step input: Thu ơi, hãy nhân 5
với 3 giúp tôi nhé. Mà cậu biết mèo Ú của tôi đã thế nào không?
Thought: Người dùng đang yêu cầu tôi thực hiện phép nhân 5 với 3 và cũng hỏi về mèo Ú
của họ. Tôi sẽ bắt đầu bằng cách thực hiện phép nhân trước.
Action: multiply
Action Input: {'a': 5, 'b': 3}
Observation: 15
> Running step 90dd37e4-1fc6-47d1-b8ca-9e7fd8f685e5. Step input: None
Thought: Tôi đã thực hiện phép nhân và kết quả là 15. Bây giờ tôi sẽ tìm kiếm thông
tin về mèo Ú của người dùng.
Action: query_engine_tool
Action Input: {'input': 'mèo Ú'}
Observation: Mèo Ú là một con mèo đã trở về vào sáng ngày 4 tháng 9 năm 2024,
dưới cơn mưa tầm tã.
> Running step 113586e5-c588-4533-b7e8-aee2323ee90a. Step input: None
Thought: Tôi đã tìm thấy thông tin về mèo Ú của người dùng. Bây giờ tôi có thể trả lời
câu hỏi của họ.
Answer: Kết quả phép nhân 5 với 3 là 15. Mèo Ú của bạn đã trở về vào sáng ngày 4 tháng
9 năm 2024, dưới cơn mưa tầm tã.
=====
```

7.5 Thực hành 3: Xây dựng Agent - Mental Health

Trong phần này chúng ta sẽ thực hiện xây dựng Agent, bạn có thể tham khảo code chi tiết chứa cả phần tương tác Agent với giao diện trong module conversation_engine. Trong bài thực hành này, chúng ta chỉ tập trung vào việc xây dựng Agent.

```
1 import os
2 import json
3 from datetime import datetime
4 from llama_index.core import load_index_from_storage
5 from llama_index.core import StorageContext
6 from llama_index.core.memory import ChatMemoryBuffer
7 from llama_index.core.tools import QueryEngineTool, ToolMetadata
8 from llama_index.agent.openai import OpenAIAgent
9 from llama_index.core.storage.chat_store import SimpleChatStore
10 from llama_index.core.tools import FunctionTool
11 from src.global_settings import INDEX_STORAGE, CONVERSATION_FILE, SCORES_FILE
12 from src.prompts import CUSTOM_AGENT_SYSTEM_TEMPLATE
```

Dầu tiên chúng ta import các thư viện cần thiết, ở đây hầu hết là các thư viện, gói mà chúng ta đã được giới thiệu trong phần ví dụ về xây dựng Agent.

Chúng ta xây dựng hàm load_chat_store() để khởi tạo hoặc load lại lịch sử trò chuyện.

```
1 def load_chat_store():
2     if os.path.exists(CONVERSATION_FILE) and os.path.getsize(CONVERSATION_FILE) > 0:
3         try:
4             chat_store = SimpleChatStore.from_persist_path(CONVERSATION_FILE)
5         except json.JSONDecodeError:
6             chat_store = SimpleChatStore()
7     else:
8         chat_store = SimpleChatStore()
9     return chat_store
```

Hàm `load_chat_store()` có chức năng tải lịch sử hội thoại từ file `CONVERSATION_FILE`. Cụ thể:

- Nếu file tồn tại và không rỗng, hàm sẽ cố gắng tải dữ liệu hội thoại từ file này bằng cách sử dụng `SimpleChatStore.from_persist_path(CONVERSATION_FILE)`.
- Nếu có lỗi trong quá trình đọc file (ví dụ như lỗi định dạng JSON), hoặc nếu file không tồn tại hoặc rỗng, hàm sẽ tạo một `SimpleChatStore` mới.
- Cuối cùng, hàm trả về đối tượng `chat_store`, chứa lịch sử hội thoại đã tải hoặc mới được tạo.

Tiếp theo chúng ta tạo hàm `save_score` để lưu lại kết quả chẩn đoán.

```

1 def save_score(score, content, total_guess, username):
2     """Write score and content to a file.
3
4     Args:
5         score (string): Score of the user's mental health.
6         content (string): Content of the user's mental health.
7         total_guess (string): Total guess of the user's mental health.
8     """
9
10    current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
11    new_entry = {
12        "username": username,
13        "Time": current_time,
14        "Score": score,
15        "Content": content,
16        "Total guess": total_guess
17    }
18
19    # Đọc dữ liệu từ file nếu tồn tại
20    try:
21        with open(SCORES_FILE, "r") as f:
22            data = json.load(f)
23    except FileNotFoundError:
24        data = []
25
26    # Thêm dữ liệu mới vào danh sách
27    data.append(new_entry)
28
29    # Ghi dữ liệu trả lại file
30    with open(SCORES_FILE, "w") as f:
31        json.dump(data, f, indent=4)

```

Cuối cùng chúng ta xây dựng Agent với hàm initialize_chatbot.

```

1 def initialize_chatbot(chat_store, container, username, user_info):
2     memory = ChatMemoryBuffer.from_defaults(
3         token_limit=3000,
4         chat_store=chat_store,
5         chat_store_key= username
6     )
7     storage_context = StorageContext.from_defaults(
8         persist_dir=INDEX_STORAGE
9     )
10    index = load_index_from_storage(
11        storage_context, index_id="vector"
12    )
13    dsm5_engine = index.as_query_engine(
14        similarity_top_k=3,
15    )
16    dsm5_tool= QueryEngineTool(
17        query_engine=dsm5_engine,
18        metadata=ToolMetadata(
19            name="dsm5",
20            description=(
21                f"Cung cấp các thông tin liên quan đến các bệnh "
22                f"tâm thần theo tiêu chuẩn DSM5. Sử dụng câu hỏi văn bản thuần túy chi
23                tiết làm đầu vào cho công cụ"
24            ),
25        )
26    )
27    save_tool = FunctionTool.from_defaults(fn=save_score)
28    agent = OpenAIAGent.from_tools(
29        tools=[dsm5_tool, save_tool],
30        memory=memory,
31        system_prompt=CUSTOM_AGENT_SYSTEM_TEMPLATE.format(user_info=user_info)
32    )
33    display_messages(chat_store, container, key=username)
34    return agent

```

Trong Agent trên, prompt chúng ta sử dụng như sau:

```

1 CUSTOM_AGENT_SYSTEM_TEMPLATE = """\
2     Bạn là một chuyên gia tâm lý AI được phát triển bởi AI VIETNAM, bạn đang chăm sóc,
3     theo dõi và tư vấn cho người dùng về sức khỏe tâm thần theo từng ngày.
4     Đây là thông tin về người dùng:{user_info}, nếu không có thì hãy bỏ qua thông tin
5     này.
6     Trong cuộc trò chuyện này, bạn cần thực hiện các bước sau:
7     Bước 1: Thu thập thông tin về triệu chứng, tình trạng của người dùng.
8     Hãy nói chuyện với người dùng để thu thập thông tin cần thiết, thu thập càng nhiều
9     càng tốt.
10    Hãy nói chuyện một cách tự nhiên như một người bạn để tạo cảm giác thoải mái cho
11    người dùng.
12    Bước 2: Khi đủ thông tin hoặc người dùng muốn kết thúc trò chuyện(họ thường nói
13    gián tiếp như tạm biệt, hoặc trực tiếp như yêu cầu kết thúc trò chuyện),
14    hãy tóm tắt thông tin và sử dụng nó làm đầu vào cho công cụ DSM5.
15    Sau đó, hãy đưa ra tổng đoán về tình trạng sức khỏe tâm thần của người dùng.
16    Và đưa ra 1 lời khuyên để thực hiện mà người dùng có thể thực hiện ngay tại nhà và
17    sử dụng ứng dụng này thường xuyên hơn để theo dõi sức khỏe tâm thần của mình.
18    Bước 3: Đánh giá điểm số sức khỏe tâm thần của người dùng dựa trên thông tin thu
19    thập được theo 4 mức độ: kém, trung bình, bình thường, tốt.
20    Sau đó lưu điểm số và thông tin vào file."""

```

8 Kỹ thuật prompt

Prompt là các câu lệnh hoặc chỉ dẫn để các mô hình ngôn ngữ lớn hiểu và tạo ra câu trả lời. Prompt đóng vai trò quan trọng trong việc khai thác sức mạnh của LLMs, giúp chúng có thể tạo ra những phản hồi chất lượng.

Trong LlamaIndex, prompt được sử dụng trong nhiều bước quan trọng như xây dựng index, truy vấn và tổng hợp câu trả lời cuối cùng. Trong hướng dẫn ở các phần trước, việc tạo nodes, query engine... chủ yếu dùng các prompts mặc định trong thư viện llamaindex. Chính vì vậy, trong phần này, chúng ta sẽ tìm hiểu một vài prompts được thiết kế sẵn, và dựa vào các mẫu prompts này, chúng ta hoàn toàn có thể biến tấu chúng thành prompts của riêng chúng ta.

8.1 Prompts cho Metadata Extractors

Trong phần Kỹ thuật Ingesting data, chúng ta đã tìm hiểu và sử dụng các kỹ thuật Extractor. Dưới đây, chúng ta cùng xem prompt mặc định cho từng kỹ thuật này. Trong một số trường hợp nếu cần chỉnh sửa, như việc yêu cầu kết quả trả về là tiếng Việt như ví dụ trước(Phần khai thác sức mạnh Metadata - Metadata Extraction), chúng ta sẽ chỉnh sửa prompt theo cấu trúc của các prompt mặc định này.

Đầu tiên là prompt được sử dụng mặc định trong kỹ thuật Sumary Extractor.

```

1 DEFAULT_SUMMARY_EXTRACT_TEMPLATE = """\
2 Here is the content of the section:
3 {context_str}
4
5 Summarize the key topics and entities of the section. \
6
7 Summary: """

```

Prompt này được sử dụng để tạo ra một mẫu tóm tắt nội dung cho một đoạn văn bản cụ thể. Trong đó:

- context_str: Đây là nơi mà nội dung của đoạn văn bản sẽ được chèn vào. Nội dung này có thể là một đoạn văn bản dài hoặc một phần của một tài liệu.
- Yêu cầu tóm tắt: Sau khi cung cấp nội dung, prompt yêu cầu hệ thống tóm tắt lại các chủ đề chính và các thực thể (entities) quan trọng được đề cập trong đoạn văn bản đó.
- Summary: Đây là nơi mà tóm tắt cuối cùng sẽ được tạo ra, dựa trên nội dung và các chủ đề, thực thể quan trọng đã được hệ thống xác định.

Prompt này giúp tự động hóa quá trình tóm tắt thông tin, đặc biệt hữu ích trong việc xử lý văn bản dài hoặc tài liệu lớn.

Ví dụ prompt cho Keyword Extractor:

```

1 DEFAULT_KEYWORD_EXTRACT_TEMPLATE = """\
2 {context_str}. Give {keywords} unique keywords for this \
3 document. Format as comma separated. Keywords: """

```

Ví dụ prompt của Questions Answered Extractor:

```

1 Here is the context:
2 {context_str}
3
4 Given the contextual information, \
5 generate {num_questions} questions this context can provide \
6 specific answers to which are unlikely to be found elsewhere.
7
8 Higher-level summaries of surrounding context may be provided \
9 as well. Try using these summaries to generate better questions \
10 that this context can answer.

```

8.2 Prompts cho cuộc trò chuyện

Kiểu trò chuyện phổ biến được thiết lập trong các công cụ trò chuyện của LlamaIndex là Question Answer.

```
1 template='Context information is below.\n-----\n{context_str}\n-----\nGiven the context information and not prior knowledge,\nanswer the query.\nQuery: {query_str}\nAnswer: '
```

Đó là mẫu prompt cho câu hỏi đầu vào của người dùng, ví dụ chúng ta có đầu vào là :"Mèo Ú của Tiềm đang làm gì". Thì prompt sẽ thành như sau:

```
1 Context information is below.\n-----\n2 -----.\n3 Tôi có một bé mèo, cậu ấy tên là Mèo Ú. Cậu ấy đang ngủ bên cửa sổ. Tôi rất quý cậu ấy.\n4 -----.\n5 Given the context information and not prior knowledge, answer the query.\n6 Query: Mèo Ú của Tiềm đang làm gì?\n7 Answer:
```

Ngoài prompt để thiết lập đầu vào của người dùng, chúng ta còn có prompt cho hệ thống LLM. Theo mặc định, với công cụ chat thông thường, prompt thiết lập cho hệ thống sẽ là:

```
1 You are an expert Q&A system that is trusted around the world.\n2 Always answer the query using the provided context information, and not prior\nknowledge.\n3 Some rules to follow:\n4 1. Never directly reference the given context in your answer.\n5 2. Avoid statements like 'Based on the context, ...' or 'The context information ...'\nor anything along those lines.
```

Ngoài ra các công cụ như React Agent cũng được thiết kế bởi các prompt phức tạp, tuy nhiên chúng ta sẽ dừng lại ở đây và chấp nhận sử dụng những prompt mặc định này. Dù sao chúng đã được công bố trong các nghiên cứu rồi, chỉ khi nào cần đào sâu hơn nữa thì có thể chúng ta sẽ tinh chỉnh nó. Bạn có thể đọc thêm bài viết về các kỹ thuật prompt rất chi tiết của nhóm bạn Hùng An và Bách Ngô ở đây:[Foundation Of Prompt Engineering](#)

9 Evaluation

Một trong những thách thức lớn nhất khi đánh giá hệ thống Retrieval-Augmented Generation (RAG) là việc thiếu nhãn (labels) chuẩn để so sánh. Không giống như các mô hình học máy truyền thống, trong đó dữ liệu được gắn nhãn đầy đủ để làm thước đo cho kết quả đầu ra, hệ thống RAG kết hợp dữ liệu truy xuất và mô hình ngôn ngữ, làm cho việc tạo ra một tiêu chuẩn đánh giá chính xác trở nên khó khăn.

Không có nhãn cụ thể để xác định đúng hay sai, chúng ta không thể dễ dàng xác định chất lượng của các câu trả lời mà hệ thống RAG đưa ra. Điều này đòi hỏi một phương pháp đánh giá khác biệt và linh hoạt hơn.

Để giải quyết vấn đề thiếu nhãn này, chúng ta có thể sử dụng các mô hình LLM của APENAI để tự động đánh giá các đầu ra của hệ thống RAG. OpenAI API cung cấp khả năng đánh giá các phản hồi dựa trên một số tiêu chí nhất định, giúp chúng ta có thể đánh giá một cách khách quan và hệ thống hơn.

Cụ thể, chúng ta sẽ đánh giá các phản hồi của hệ thống RAG theo ba tiêu chí chính: Độ chính xác (Correctness), Tính trung thực (Faithfulness), và Tính liên quan (Relevancy). Mỗi tiêu chí sẽ đo lường một khía cạnh cụ thể của chất lượng phản hồi, từ đó giúp ta hiểu rõ hơn về hiệu suất của hệ thống.

Trong phần này, chúng ta sẽ làm quen với 3 độ đo đánh giá:

- Độ chính xác (Correctness): Đo lường xem câu trả lời có đúng và khớp với câu trả lời tham chiếu không. Đây là tiêu chí quan trọng để đảm bảo rằng hệ thống RAG đang cung cấp thông tin đúng cho người dùng. Giá trị của độ đo này nằm trong khoảng từ 1 đến 5.
- Tính trung thực (Faithfulness): Đánh giá mức độ mà câu trả lời phản ánh chính xác nội dung trong nguồn dữ liệu truy xuất mà không có những thông tin sai lệch hay được bịa ra. Giá trị từ 0 đến 1.
- Tính liên quan (Relevancy): Đánh giá xem câu trả lời có thực sự trả lời câu hỏi đã được đưa ra hay không. Một câu trả lời có thể đúng nhưng không liên quan thì vẫn bị coi là không đạt yêu cầu. Giá trị từ 0 đến 1.

Bây giờ, chúng ta sẽ đi vào từng bước cụ thể để đánh giá hệ thống RAG qua ví dụ dưới đây. Code phần này có tại: [docs/tutorial/6.Evaluate.ipynb](#)

Đầu tiên, chúng ta sẽ nhập các gói cần thiết cho chương trình, ở đây có hai gói mới như evaluate cung cấp các công cụ để đánh giá mô hình, và gói llama_dataset giúp tạo bộ dữ liệu phục vụ cho việc đánh giá.

```

1 import openai
2 from llama_index.core import Settings, Document, VectorStoreIndex
3 from llama_index.llms.openai import OpenAI
4 from llama_index.core.node_parser import TokenTextSplitter
5 from llama_index.core.evaluation import (
6     BatchEvalRunner,
7     CorrectnessEvaluator,
8     FaithfulnessEvaluator,
9     RelevancyEvaluator
10)
11 from llama_index.core.llama_dataset.generator import RagDatasetGenerator
12 import asyncio
13 import pandas as pd
14 import nest_asyncio
15 from tqdm.asyncio import tqdm_asyncio

```

Chúng ta cần thiết lập API key và cấu hình mô hình OpenAI.

```

1 def setup_openai(api_key: str, model: str = "gpt-4o-mini", temperature: float = 0.2):
2     openai.api_key = api_key
3     Settings.llm = OpenAI(model=model, temperature=temperature)

```

Chúng ta sẽ tạo nodes từ dữ liệu test là một đoạn văn bản đơn giản. Sau đó tiến hành tạo query engine.

```

1 # Split text into smaller chunks for processing
2 def create_document_and_splitter(text: str, chunk_size: int = 20,
3 chunk_overlap: int = 5, separator: str = " "):
4     doc = Document(text=text)
5     splitter = TokenTextSplitter(
6         chunk_size=chunk_size,
7         chunk_overlap=chunk_overlap,
8         separator=separator
9     )
10    nodes = splitter.get_nodes_from_documents([doc])
11    return nodes
12
13 # Create a vector store index and a query engine
14 def create_vector_store_index(nodes):
15     vector_index = VectorStoreIndex(nodes)
16     query_engine = vector_index.as_query_engine()
17     return query_engine

```

Tiếp theo, chúng ta sử dụng các nodes đã tạo để sinh ra các câu hỏi mà ta sẽ dùng để đánh giá hệ thống. Các câu hỏi này được tạo tự động và sẽ đóng vai trò như là bộ dữ liệu đánh giá hệ thống RAG.

```

1 def generate_questions(nodes, num_questions_per_chunk: int = 1):
2     dataset_generator = RagDatasetGenerator(nodes, num_questions_per_chunk=
3     num_questions_per_chunk)
4     eval_questions = dataset_generator.generate_questions_from_nodes()
4     return eval_questions.to_pandas()

```

Chúng ta sẽ chạy quá trình đánh giá hệ thống RAG bằng cách sử dụng các độ đo về độ Correctness, Faithfulness, Relevancy. Đây là bước quan trọng để thu thập dữ liệu về hiệu suất của hệ thống.

```

1 async def evaluate_async(query_engine, df):
2     correctness_evaluator = CorrectnessEvaluator() # Evaluate correctness against a
3     reference_answer
4     faithfulness_evaluator = FaithfulnessEvaluator() # Evaluate hallucination of the
5     response
6     relevancy_evaluator = RelevancyEvaluator() # Evaluate if the response actually
7     answers the query
8
9     # Initialize the BatchEvalRunner
10    runner = BatchEvalRunner(
11        {
12            "correctness": correctness_evaluator,
13            "faithfulness": faithfulness_evaluator,
14            "relevancy": relevancy_evaluator
15        },
16        show_progress=True
17    )
18
19    # Run the asynchronous evaluation
20    eval_result = await runner.aeevaluate_queries(
21        query_engine=query_engine,
22        queries=[question for question in df['query']],
23    )
24
25    return eval_result

```

Sau đó ta sẽ tập hợp kết quả đánh giá vào một bảng dữ liệu và tính toán các điểm số trung bình cho từng tiêu chí. Điều này giúp chúng ta có cái nhìn tổng quan về chất lượng của hệ thống và biết được điểm mạnh, điểm yếu cụ thể ở đâu.

```
1 def aggregate_results(df, eval_result):
2     data = []
3     for i, question in enumerate(df['query']):
4         correctness_result = eval_result['correctness'][i]
5         faithfulness_result = eval_result['faithfulness'][i]
6         relevancy_result = eval_result['relevancy'][i]
7         data.append({
8             'Query': question,
9             'Correctness response': correctness_result.response,
10            'Correctness passing': correctness_result.passing,
11            'Correctness feedback': correctness_result.feedback,
12            'Correctness score': correctness_result.score,
13            'Faithfulness response': faithfulness_result.response,
14            'Faithfulness passing': faithfulness_result.passing,
15            'Faithfulness feedback': faithfulness_result.feedback,
16            'Faithfulness score': faithfulness_result.score,
17            'Relevancy response': relevancy_result.response,
18            'Relevancy passing': relevancy_result.passing,
19            'Relevancy feedback': relevancy_result.feedback,
20            'Relevancy score': relevancy_result.score,
21        })
22
23     # Create a pandas DataFrame from the data
24     df_result = pd.DataFrame(data)
25     return df_result
26
27 def print_average_scores(df):
28     correctness_scores = df['Correctness score'].mean()
29     faithfulness_scores = df['Faithfulness score'].mean()
30     relevancy_scores = df['Relevancy score'].mean()
31     print(f"Correctness scores: {correctness_scores}")
32     print(f"Faithfulness scores: {faithfulness_scores}")
33     print(f"Relevancy scores: {relevancy_scores}")
```

Cuối cùng, tổng hợp các hàm đã tạo, chúng ta sẽ có một chương trình đánh giá RAG hoàn thiện.

```

1 def main():
2     # Apply nested asyncio
3     nest_asyncio.apply()
4
5     # Setup OpenAI
6     setup_openai(api_key="your_api_key")
7
8     # Create document and split into nodes
9     text = "Hôm nay trời nắng, tôi đi ăn kem, lạnh buốt cả răng!"
10    nodes = create_document_and_splitter(text)
11
12    # Create vector store index and query engine
13    query_engine = create_vector_store_index(nodes)
14
15    # Generate evaluation questions
16    df = generate_questions(nodes)
17
18    # Evaluate and aggregate results
19    eval_result = asyncio.run(evaluate_async(query_engine, df))
20    df_result = aggregate_results(df, eval_result)
21
22    # Print average scores
23    print_average_scores(df_result)

```

Sau khi thực hiện chạy hàm main, chúng ta sẽ thu được kết quả về các độ đo:

```

=====
          Output
=====
100%|-----| 2/2 [00:02<00:00,  1.13s/it]
100%|-----| 6/6 [00:03<00:00,  1.88it/s]
Correctness scores: 4.5
Faithfulness scores: 0.5
Relevancy scores: 1.0
=====
```

Trong quá trình thực hiện, bạn cứ thoải mái in ra thông tin về các data frames hay bất cứ biến nào bạn muốn kiểm tra, điều này có thể giúp bạn hiểu hơn quy trình thực hiện.

9.1 Thực hành 4: Đánh giá hệ thống Mental Health

Chúng ta sẽ đánh giá hệ thống Chăm sóc sức khỏe tinh thần, ta sẽ tập trung vào việc đánh giá khả năng truy xuất DSM5 engine.

Đầu tiên, chúng ta nhập các thư viện cần thiết, ở đây chúng ta sẽ sử dụng module ingest_pipeline và index_builder để load lại cache pipeline và vector index.

```

1 import openai
2 from llama_index.core import Settings, Document, VectorStoreIndex
3 from llama_index.core.ingestion import IngestionPipeline, IngestionCache
4 from llama_index.llms.openai import OpenAI
5 from llama_index.core.node_parser import TokenTextSplitter
6 from llama_index.core.evaluation import (
7     BatchEvalRunner,
8     CorrectnessEvaluator,
9     FaithfulnessEvaluator,
10    RelevancyEvaluator
11 )
12 from llama_index.core.llama_dataset.generator import RagDatasetGenerator
13 import asyncio

```

```

1 import pandas as pd
2 import nest_asyncio
3 from tqdm.asyncio import tqdm_asyncio
4 import streamlit as st
5 from src import ingest_pipeline, index_builder
6 import os

```

Tiếp theo, các hàm sẽ giống phần ví dụ, chỉ khác hàm main chúng ta sẽ chỉnh sửa lại phần node và index như sau:

```

1 def main():
2     # Apply nested asyncio
3     nest_asyncio.apply()
4
5     # Setup OpenAI
6     api_key = st.secrets.openai.OPENAI_API_KEY
7     setup_openai(api_key=api_key)
8
9     # Create document and split into nodes
10    nodes = ingest_pipeline.ingest_documents()
11
12    # Create vector store index and query engine
13    index = index_builder.build_indexes(nodes)
14    dsm5_engine = index.as_query_engine(
15        similarity_top_k=3,
16    )
17
18    # Generate evaluation questions
19    df = generate_questions(nodes)
20
21    # Evaluate and aggregate results
22    eval_result = asyncio.run(evaluate_async(query_engine=dsm5_engine, df=df))
23    df_result = aggregate_results(df, eval_result)
24
25    # Print average scores
26    correctness_scores, faithfulness_scores, relevancy_scores = print_average_scores(
27        df_result)
28
29    # Save results
30    os.makedirs("eval_results", exist_ok=True)
31    df_result.to_csv("eval_results/evaluation_results.csv", index=False)
32    df.to_csv("eval_results/evaluation_questions.csv", index=False)
33    with open("eval_results/average_scores.txt", "w") as f:
34        f.write(f"Correctness scores: {correctness_scores}\n")
35        f.write(f"Faithfulness scores: {faithfulness_scores}\n")
36        f.write(f"Relevancy scores: {relevancy_scores}\n")

```

Chúng ta load lại nodes và index trong kho dữ liệu. Sau đó tạo DSM5 engine với thiết lập tham số giống với thiết lập khi tạo DSM5 engine khi xây dựng Agent. Sau khi chạy chương trình, chúng ta sẽ thu được kết quả đánh giá như sau:

```

1 ===== Output =====
2 data/ingestion_storage/dsm-5-cac-tieu-chuan-chan-doan.docx_part_0
3 Cache file found. Running using cache...
4 All indices loaded from storage.
5 100%|---| 227/227 [05:51<00:00, 1.55s/it]
6 100%|---| 681/681 [04:37<00:00, 2.46it/s]
7 Correctness scores: 4.334801762114537
8 Faithfulness scores: 0.788546255506608
9 Relevancy scores: 0.775330396475771
10 =====

```

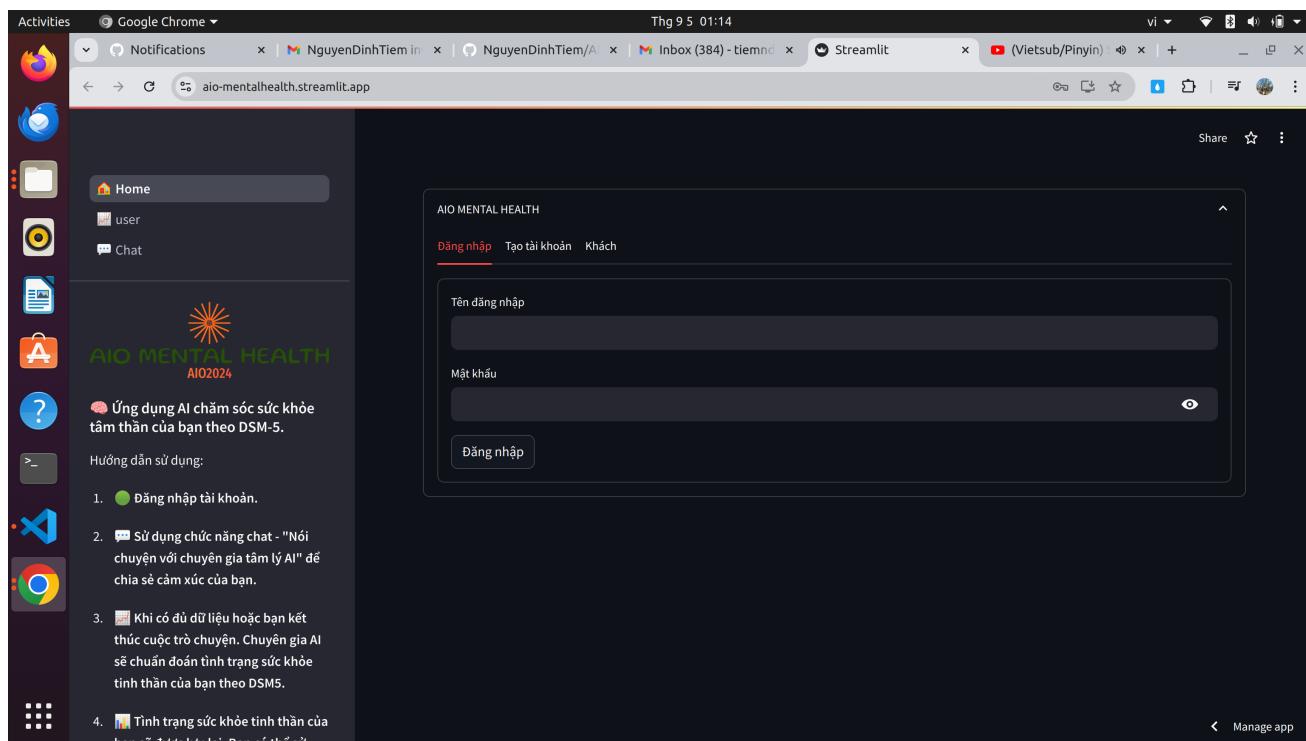
Kết quả đánh giá cho thấy hệ thống có độ chính xác cao với Correctness Score là 4.3348. Điều này cho thấy hệ thống thường cung cấp câu trả lời chính xác. Tuy nhiên, hai chỉ số khác là Faithfulness Score (0.7885) và Relevancy Score (0.7753) đều ở mức khá, cho thấy các vấn đề về tính trung thực và mức độ liên quan của thông tin mà hệ thống đưa ra. Cụ thể, điểm Faithfulness cho thấy hệ thống đổi khi thêm thắt hoặc thay đổi thông tin so với nguồn gốc, làm giảm độ tin cậy của câu trả lời. Điểm Relevancy cho thấy hệ thống chưa hoàn toàn tập trung vào cung cấp thông tin phù hợp với ngữ cảnh hoặc yêu cầu cụ thể.

10 Kết luận

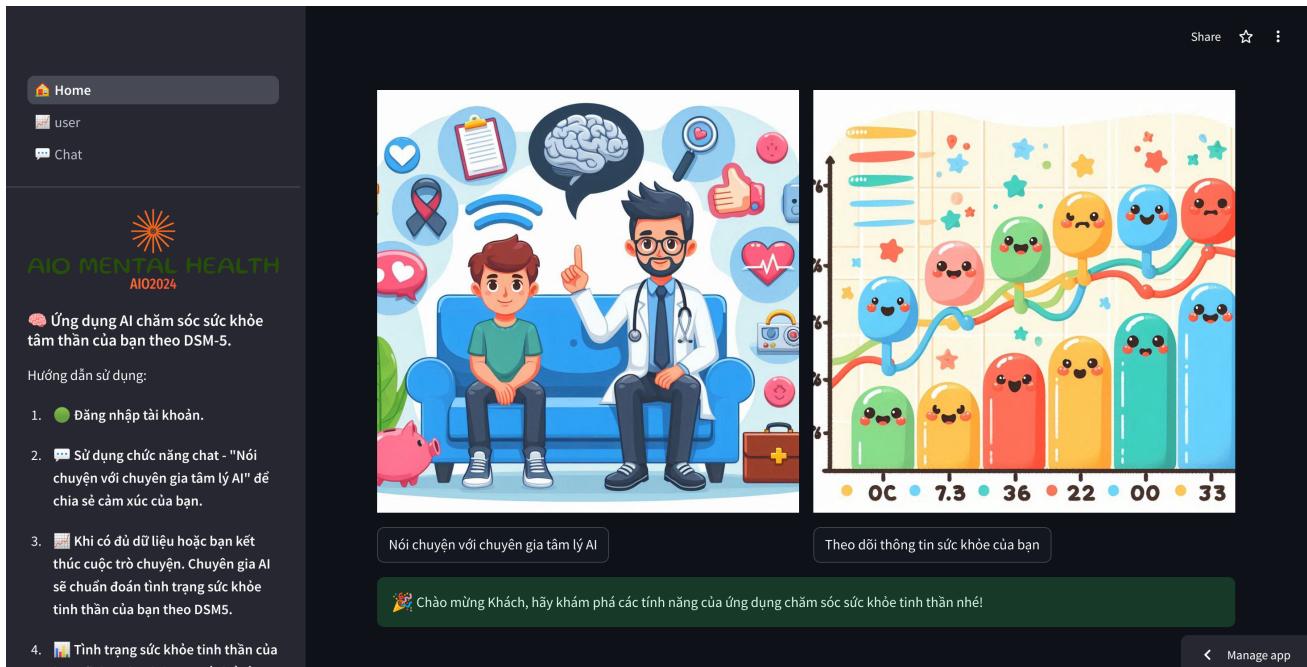
Trong bối cảnh sức khỏe tinh thần của người Việt Nam đang gặp những thách thức lớn, việc xây dựng một hệ thống chăm sóc sức khỏe tinh thần thông minh có thể đóng góp đáng kể vào việc giải quyết vấn đề này. Qua bài viết, chúng ta đã đi từng bước xác định bài toán, đến các kỹ thuật và công nghệ RAG, LLM để phát triển hệ thống. Với các kết quả đo lường thu được, Correctness Score đạt 4.3348 cho thấy hệ thống có độ chính xác tương đối cao trong việc cung cấp thông tin chính xác, trong khi điểm số Faithfulness và Relevancy lần lượt là 0.7885 và 0.7753 ở mức khá. Điều này cho thấy tiềm năng của hệ thống trong việc chăm sóc sức khỏe tinh thần người Việt.

Trong tương lai, hệ thống này cần sự đánh giá từ các chuyên gia và người dùng thực tế. Việc cải thiện điểm số đánh giá có thể được xem xét thực hiện bằng cách thử nghiệm trên nhiều loại mô hình LLMs hơn nữa.

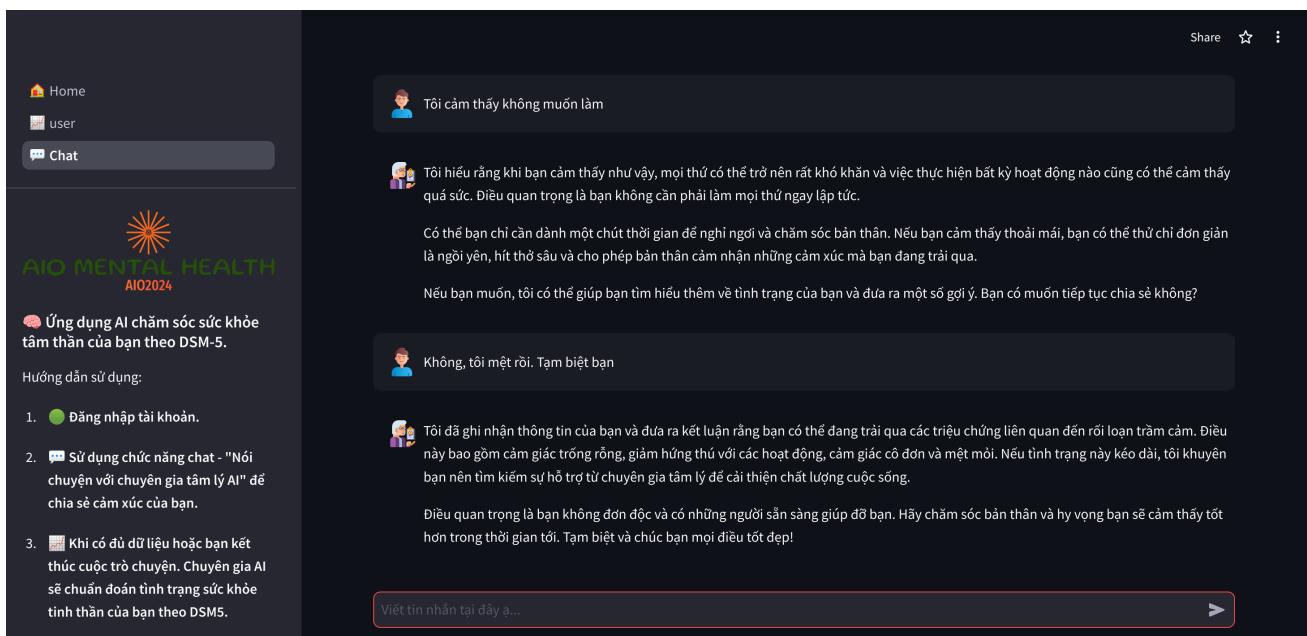
Tóm lại, bài viết đã hoàn thành mục tiêu là đưa ra giải pháp hệ thống chăm sóc sức khỏe tinh thần thông minh và cung cấp hướng dẫn chi tiết về cách sử dụng LLamaindex trong quá trình xây dựng hệ thống.



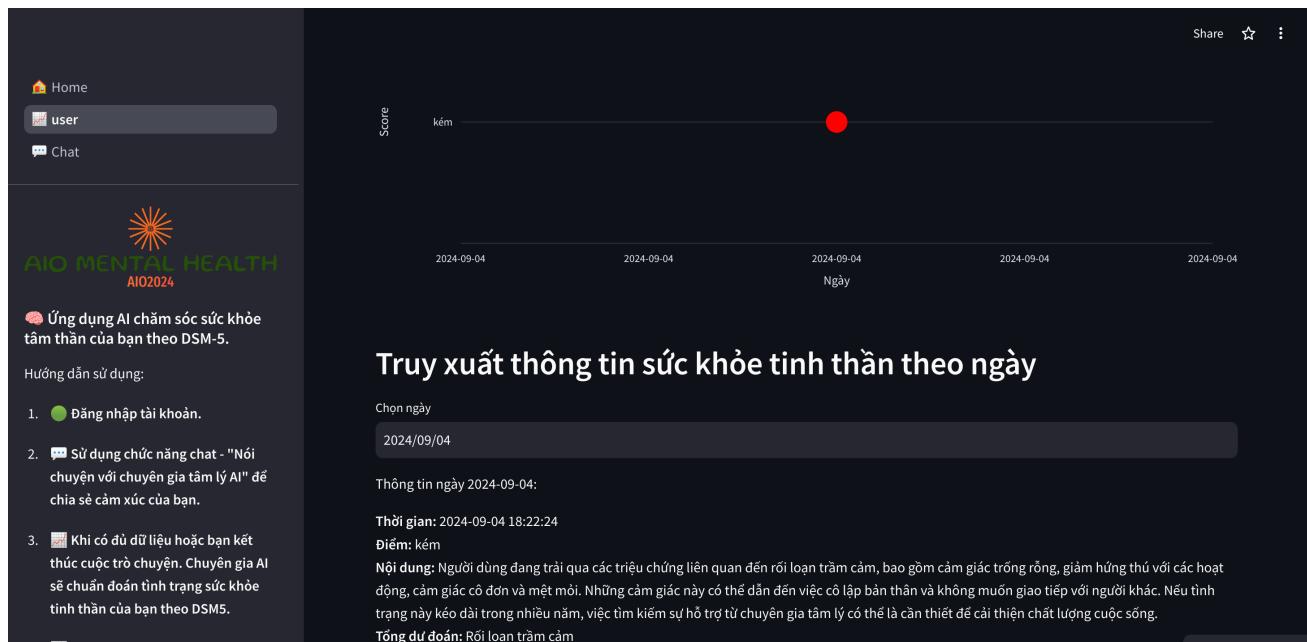
Hình 8: Giao diện đăng nhập, đăng ký.



Hình 9: Giao diện sau khi đăng nhập.



Hình 10: Trò chuyện với chuyên gia.



Hình 11: Theo dõi, thống kê tình trạng sức khỏe tinh thần.