RAG với dữ liệu phi cấu trúc

RAG (Retrieval-Augmented Generation) là một phương pháp tiên tiến kết hợp giữa truy xuất thông tin và sinh ngôn ngữ tự nhiên, đặc biệt hiệu quả khi làm việc với dữ liệu phi cấu trúc. Thay vì chỉ dựa vào dữ liệu huấn luyện sẵn có, RAG chủ động tìm kiếm và khai thác thông tin từ các nguồn bên ngoài để tạo ra câu trả lời chính xác và giàu ngữ cảnh hơn. Điều này đặc biệt hữu ích trong các tình huống mà dữ liệu không được tổ chức theo cấu trúc rõ ràng, như văn bản tự do, tài liệu PDF hoặc nội dung web.

- Khai phá dữ liệu phi cấu trúc với Retrieval-Augmented Generation (RAG)
- Giới thiệu về Thư viện LangChain – Chìa khóa để xử lý dữ liệu phi cấu trúc
- Xử lý File Excel Không Cấu Trúc với LangChain
- Thiết lập môi trường xử lý dữ liệu không có cấu trúc với LangChain
- Đọc và xử lý dữ liệu Excel với LangChain
- Xây dựng Hệ thống Truy xuất Thông tin với LangChain + OpenAI
- Xây dựng hệ thống RAG với LangChain và OpenAI

Khai phá dữ liệu phi cấu trúc với Retrieval-Augmented Generation (RAG)

Hãy tưởng tượng bạn đang phải đối mặt với hàng tá tài liệu, báo cáo dài dòng, hợp đồng, email, hay các tệp PDF, Word, PowerPoint… và bạn chỉ cần một mảnh thông tin nhỏ ẩn sâu bên trong đó. Việc tìm kiếm, tóm tắt hoặc phân tích những dữ liệu này theo cách thủ công thật sự mất thời gian và gây mệt mỏi.

Đây chính là thách thức của dữ liệu phi cấu trúc – những loại dữ liệu không tuân theo định dạng hàng cột quen thuộc như trong cơ sở dữ liệu hay bảng tính. Và đó cũng là lý do chúng ta cần đến RAG.

RAG (Retrieval-Augmented Generation) là giải pháp giúp bạn:

- **Tự động truy xuất thông tin** từ các tệp không có cấu trúc.
- **Tóm tắt nội dung** một cách thông minh.
- **Trả lời câu hỏi** dựa trên ngữ cảnh thực tế từ dữ liệu bạn cung cấp.

Trong chương này, bạn sẽ:
Làm quen với thư viện **LangChain** – công cụ chính giúp xử lý tài liệu phi cấu trúc.
Học cách xử lý nhiều loại dữ liệu: Excel, Word, PowerPoint, PDF, EPUB…
Xây dựng hệ thống **retrieval** từ các tài liệu này.
Tùy chỉnh các hàm để **truy xuất và sinh câu trả lời có ý nghĩa từ dữ liệu thực tế**.

Kết quả sau chương học:
Bạn sẽ sở hữu **bộ công cụ hoàn chỉnh** để làm việc với dữ liệu phi cấu trúc: từ việc tải, chia nhỏ văn bản, đến truy xuất thông tin và tạo nội dung có giá trị. Tất cả được áp dụng trong các **bài tập thực tế, nhiều coding và ví dụ minh họa rõ ràng**.

> **Dữ liệu phi cấu trúc có mặt ở khắp nơi – email, báo cáo, mạng xã hội, sách điện tử… và khả năng khai thác nó không chỉ là kỹ năng kỹ thuật, mà là một siêu năng lực.**
> Hãy cùng bắt đầu hành trình này!

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Giới thiệu về Thư viện LangChain – Chìa khóa để xử lý dữ liệu phi cấu trúc

Vì sao cần LangChain?

Ở phần trước, chúng ta đã sử dụng API của OpenAI để xử lý hình ảnh và văn bản. Tuy nhiên, có một vấn đề:

> Mỗi trang được xử lý riêng lẻ → Chúng ta **bỏ lỡ mối liên kết giữa các trang.**

Hãy tưởng tượng nếu điều này xảy ra với cả một **tài liệu PowerPoint, một bảng tính Excel, hoặc một cuốn sách điện tử dài hàng trăm trang**. Việc chỉ xử lý từng phần riêng biệt là **không hiệu quả**.

Vì thế, ta cần **một công cụ mạnh mẽ hơn**. Đó chính là **LangChain**.

LangChain là gì?

LangChain là một **thư viện và framework cực kỳ mạnh mẽ**, được thiết kế để:

- Xây dựng các ứng dụng sử dụng mô hình ngôn ngữ lớn (LLMs).
- Làm việc với dữ liệu **phi cấu trúc** như tài liệu, email, ebook, báo cáo, v.v.
- Tạo ra quy trình xử lý phức tạp một cách **dễ dàng, có tổ chức**.

Nói đơn giản: **LangChain giúp bạn kết nối các bước xử lý dữ liệu lại với nhau** – như một chuỗi (chain).

Tại sao nên dùng LangChain?

LangChain được xây dựng để **giúp bạn tập trung vào việc trích xuất thông tin**, thay vì viết hàng đống đoạn mã rối rắm. Nó cung cấp:

- Cách tổ chức rõ ràng các bước xử lý (load, chia nhỏ, truy vấn, sinh kết quả).
- Tích hợp tốt với các mô hình AI như **GPT-3.5, GPT-4**, và các mô hình khác.
- Hỗ trợ dữ liệu **phi cấu trúc**: Excel, Word, PowerPoint, PDF, EPUB...

Các thành phần chính trong LangChain

1. **Document Loaders**
   Dùng để tải các tài liệu từ nhiều định dạng khác nhau (PDF, DOCX, Excel,...)
2. **Text Splitters**
   Giúp chia nhỏ văn bản lớn thành từng phần nhỏ, tránh vượt quá giới hạn token của mô hình AI.
3. **Embeddings**
   Dùng để chuyển văn bản thành vector, giúp hệ thống hiểu nội dung để tìm kiếm và truy vấn.
4. **Vector Stores**
   Là nơi lưu trữ các embedding, giúp **truy xuất thông tin nhanh chóng** dựa trên nội dung.
5. **Language Models (LLMs)**
   Kết nối với mô hình như GPT để tạo câu trả lời, tóm tắt văn bản, viết lại nội dung,...

Kết nối với OpenAI thông qua LangChain

LangChain cho phép bạn cấu hình kết nối đến GPT dễ dàng. Dưới đây là các thông số cơ bản:

Tham số	Ý nghĩa
`api_key`	Khóa truy cập OpenAI của bạn
`model`	Chọn GPT-3.5, GPT-4,...
`temperature`	Mức độ sáng tạo của kết quả (0 = chính xác, 1 = sáng tạo)
`max_tokens`	Giới hạn độ dài đầu ra
`n`	Số kết quả muốn tạo ra (mặc định là 1)
`stop`	Ký tự/dấu hiệu để dừng sinh văn bản
`presence_penalty`	Phạt nếu từ đã xuất hiện trước đó (giảm lặp lại)
`frequency_penalty`	Phạt nếu từ xuất hiện nhiều lần (kiểm soát tần suất lặp)

Thực tế, bạn chỉ cần dùng **`api_key`**, **`model`**, **và** **`temperature`** là đủ để bắt đầu.

LangChain + OpenAI là **bộ công cụ mạnh mẽ** giúp bạn:

- Xử lý nhiều loại tài liệu không có cấu trúc.
- Tự động chia nhỏ và hiểu nội dung văn bản.
- Truy xuất thông tin theo yêu cầu.

- Tạo nội dung mới từ dữ liệu cũ.

Sắp tới, chúng ta sẽ đi sâu vào từng phần: từ cách tải tài liệu, chia nhỏ văn bản, đến xây dựng hệ thống tìm kiếm và sinh câu trả lời – tất cả đều thực hành, có ví dụ cụ thể.

> **Hãy chuẩn bị sẵn máy tính, trình soạn mã và tài liệu cần xử lý. Chúng ta bắt đầu hành trình cùng LangChain!**

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Xử lý File Excel Không Cấu Trúc với LangChain

Tình huống thực tế

Hãy tưởng tượng bạn được giao nhiệm vụ phân tích một **file Excel khổng lồ** chứa đầy các **phản hồi của khách hàng**.

File này có:

- Nhiều sheet,
- Nhiều dòng hữu ích,
- Nhưng cũng rất nhiều dòng rác hoặc trống.

Làm sao bạn **lọc ra thông tin quan trọng** từ một "rừng dữ liệu" như vậy?

Đây chính là lúc bạn cần đến **LangChain** – một thư viện mạnh mẽ giúp xử lý dữ liệu không có cấu trúc.

Mục tiêu của bài học:

Sau bài này, bạn sẽ biết cách:

1. Tải file Excel vào LangChain
2. Chia nhỏ dữ liệu để xử lý tốt hơn
3. Tạo **embeddings** từ dữ liệu để phục vụ cho phân tích, tìm kiếm hoặc sinh nội dung

Bước 1: Tải dữ liệu Excel

Chúng ta dùng **UnstructuredExcelLoader** trong module `langchain_community`.

Cú pháp cơ bản:

```
from langchain_community.document_loaders import UnstructuredExcelLoader


loader = UnstructuredExcelLoader("reviews.xlsx", mode="elements")
docs = loader.load()
```

`mode="elements"` giúp chia nhỏ từng phần trong file Excel.

- Nếu file của bạn cực kỳ có cấu trúc (ví dụ: bảng biểu đều đặn), bạn có thể thử `mode="table"`.
- Dữ liệu giống `pandas.read_csv()` – chỉ là cách tiếp cận "LangChain-style".

Bước 2: Chia nhỏ nội dung (Chunking)

Lý do: Các mô hình ngôn ngữ có giới hạn **số lượng token**. Nếu bạn đưa vào quá nhiều chữ, nó sẽ bị cắt mất nội dung.

Giải pháp: Dùng **RecursiveCharacterTextSplitter**

```
from langchain.text_splitter import RecursiveCharacterTextSplitter


splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=200)
chunks = splitter.split_documents(docs)
```

- **chunk_size**: độ dài mỗi đoạn văn bản.
- **chunk_overlap**: phần nội dung lặp lại giữa các đoạn (để giữ ngữ cảnh liền mạch).

Ví dụ:

- Nếu bạn đặt `chunk_size=2000` và `chunk_overlap=200`, mỗi đoạn mới sẽ giữ lại 200 ký tự cuối của đoạn trước.

Gợi ý:

- Nếu dữ liệu của bạn là đánh giá ngắn (reviews), bạn có thể giảm xuống `chunk_size=300`, `chunk_overlap=50` cho tiết kiệm tài nguyên.

Bước 3: Tạo Embeddings (biểu diễn ngữ nghĩa)

Dùng `OpenAIEmbeddings` từ LangChain để biến các đoạn văn bản thành vector ngữ nghĩa.

```
from langchain_openai import OpenAIEmbeddings


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

Mô hình `text-embedding-3-large` là phiên bản tốt nhất hiện tại của OpenAI.

Lưu ý:

- **API Key** cần được bảo mật bằng biến môi trường hoặc file `secrets`.
- Nếu bạn muốn tiết kiệm chi phí, có thể dùng model embedding nhẹ hơn hoặc tạo **embeddings tùy chỉnh** (sẽ học sau).

Tổng kết

Chúng ta đã học:

1. Cách nạp dữ liệu Excel vào LangChain
2. Cách chia nhỏ văn bản để xử lý hiệu quả
3. Cách tạo embeddings để chuẩn bị cho các tác vụ phân tích AI

Từ khóa quan trọng:

- `chunk_size` và `chunk_overlap` là 2 yếu tố then chốt ảnh hưởng đến hiệu quả và độ chính xác.

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Thiết lập môi trường xử lý dữ liệu không có cấu trúc với LangChain

Trong phần này, chúng ta sẽ **thiết lập môi trường làm việc** để xử lý **dữ liệu không có cấu trúc** bằng thư viện **LangChain** và các công cụ liên quan.

Chúng ta sẽ thực hiện điều này thông qua **Google Colaboratory**, một công cụ tuyệt vời để viết và chạy mã Python trực tuyến, đồng thời dễ dàng tích hợp với **Google Drive**.

Bước 1: Tạo môi trường làm việc trên Google Colab

- Mở thư mục `unstructured_data` trong dự án `rack`
- Nhấn chuột phải → **New More → Google Collaboratory**

Đây sẽ là nơi bạn viết các đoạn mã để xử lý dữ liệu.

Bước 2: Cài đặt các thư viện cần thiết

Thư viện chính:

- `langchain` → Xử lý dữ liệu và tạo pipeline thông minh
- `langchain-community` → Loader cho các định dạng không cấu trúc như Excel, PDF, EPUB...
- `openai` → Tạo embeddings, LLM
- `faiss-cpu` → Dùng để xây dựng hệ thống truy xuất vector (retrieval system)

```
!pip install langchain-community langchain openai faiss-cpu
```

Bước 3: Thiết lập API Key cho OpenAI

Sử dụng Google Colab, bạn có thể lưu API Key dưới dạng dữ liệu người dùng:

```
from google.colab import userdata


OPENAI_API_KEY = userdata.get('janai_course')
```

Bước 4: Kết nối với Google Drive

Google Drive sẽ là nơi chứa các file dữ liệu như `.xlsx`, `.pdf`, `.docx`, `.epub`...

```
from google.colab import drive
drive.mount('/content/drive')
```

Sau đó bạn đổi thư mục làm việc sang thư mục dữ liệu:

```
%cd /content/drive/MyDrive/your-folder-path
```

Bước 5: Import các module quan trọng

```
# Load dữ liệu Excel
from langchain_community.document_loaders import UnstructuredExcelLoader

# Chia nhỏ dữ liệu
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Gọi LLM và tạo Embeddings
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

# Vector store - FAISS
from langchain.vectorstores.faiss import FAISS

# Hiển thị Markdown trong notebook
from IPython.display import display, Markdown
```

Tổng kết

Trong bài giảng này, bạn đã:

- Tạo Google Colab notebook cho dự án
- Cài đặt các thư viện cần thiết cho xử lý dữ liệu không cấu trúc
- Kết nối API OpenAI
- Liên kết với Google Drive để truy cập dữ liệu
- Import các thành phần chính từ LangChain

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Đọc và xử lý dữ liệu Excel với LangChain

Trong bài học này, chúng ta sẽ:

1. Đọc dữ liệu từ file Excel
2. Phân tích và hiển thị dữ liệu
3. Chia nhỏ dữ liệu thành các **chunk** để chuẩn bị cho việc tạo **embeddings**

Bước 1: Đọc dữ liệu Excel

Chúng ta sẽ sử dụng **UnstructuredExcelLoader** từ `langchain_community` để đọc file `reviews.xlsx`. Đây là tập dữ liệu chứa các **đánh giá và bình luận** đến ngày **21 tháng 8 năm 2024**.

```
from langchain_community.document_loaders import UnstructuredExcelLoader


loader = UnstructuredExcelLoader("reviews.xlsx", mode="elements")
docs = loader.load()
```

Ghi chú: `mode="elements"` giúp tách nội dung trong Excel thành các phần tử nhỏ như từng dòng, từng ô – điều này giúp xử lý linh hoạt hơn.

Bước 2: Hiển thị dữ liệu

Hiển thị 5 phần tử đầu tiên để kiểm tra kết quả:

```
docs[:5]
```

Bạn sẽ thấy một số dữ liệu hiển thị như `TD`, `TR` – điều này thể hiện dữ liệu gốc được biểu diễn theo dạng bảng.

Bước 3: Chia nhỏ văn bản (Chunking)

Tại sao cần chunk?

- Dữ liệu lớn sẽ khó xử lý một lần
- Embedding có giới hạn độ dài token (vd: 4096 tokens)
- Chunk nhỏ giúp dễ dàng truy vấn và tìm kiếm hơn

Thực hiện chia nhỏ

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=2000,
    chunk_overlap=200
)

chunks = text_splitter.split_documents(docs)
```

Hiển thị 5 chunk đầu tiên:

```
chunks[:5]
```

Mẹo:

- `chunk_size=2000` có thể thay đổi tùy vào độ dài dữ liệu
- Nếu file rất lớn (nhiều triệu dòng), bạn nên dùng chunk nhỏ hơn hoặc chia theo nội dung logic hơn (theo tiêu đề, đoạn...)

Ghi chú về dữ liệu

Khi hiển thị chunk, có thể bạn sẽ thấy các thẻ như `<td>` hoặc `<tr>`:

- `<td>`: thể hiện ô trong bảng
- `<tr>`: thể hiện hàng trong bảng

Điều này là bình thường khi xử lý dữ liệu dạng bảng – bạn có thể lọc hoặc xử lý thêm nếu muốn dữ liệu "sạch" hơn.

Tổng kết

Bạn vừa học cách:

- Tải dữ liệu từ file Excel bằng LangChain
- Hiển thị một phần dữ liệu
- Chia dữ liệu thành các phần nhỏ để chuẩn bị cho bước tiếp theo

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Xây dựng Hệ thống Truy xuất Thông tin với LangChain + OpenAI

Trong bài học này, chúng ta sẽ:

1. Tạo **embeddings** từ dữ liệu
2. Lưu embeddings vào cơ sở dữ liệu (vector store)
3. Xây dựng một **retrieval system** – hệ thống tìm kiếm văn bản dựa trên ý nghĩa
4. Truy vấn dữ liệu bằng câu hỏi thực tế

Bước 1: Tạo Embeddings

Chúng ta sẽ dùng mô hình **OpenAI text-embedding-3-large** để chuyển các chunk văn bản thành vector số.

```
from langchain_openai import OpenAIEmbeddings


embeddings = OpenAIEmbeddings(
    model="text-embedding-3-large",
    openai_api_key="YOUR_API_KEY"
)
```

> **Lưu ý:** Tên model cần có dấu gạch ngang (`-`) thay vì dấu gạch dưới (`_`)
> Sai: `"text_embedding_3_large"` → Đúng: `"text-embedding-3-large"`

Bước 2: Tạo Vector Store (Database)

Chúng ta lưu các embeddings vào một cơ sở dữ liệu để có thể tìm kiếm lại.

```
from langchain.vectorstores import FAISS


db = FAISS.from_documents(chunks, embeddings)
```

Ghi chú: `FAISS` là một thư viện nhanh và hiệu quả để tìm kiếm vector tương tự.

Bước 3: Truy vấn hệ thống

Bây giờ chúng ta có thể bắt đầu truy vấn hệ thống:

```
query = "Give me my worst reviews"
results = db.similarity_search_with_score(query, k=5)
```

- `k=5`: tìm 5 đoạn văn bản gần nhất với câu hỏi
- Sử dụng **cosine similarity** để đo độ gần giữa vectors

> **Cosine Similarity:** đo góc giữa hai vector – càng gần nhau, góc càng nhỏ → văn bản càng liên quan

Kết quả Truy vấn

Kết quả sẽ là danh sách các đoạn văn bản giống với truy vấn:

```
for doc, score in results:
    print(doc.page_content, "\nScore:", score)
```

Bạn có thể thấy dữ liệu có thể còn lộn xộn (ví dụ có `<td>`, `<tr>`), điều này sẽ được cải thiện bằng bước xử lý sau.

Giải thích thêm

- Mỗi chunk có độ dài khoảng `2000 tokens` với `200 tokens` trùng lặp giữa các chunk
- Embedding giúp "mã hóa ý nghĩa" của văn bản thành vector
- Truy vấn sẽ được ánh xạ sang vector và so sánh với các chunk đã mã hóa

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Xây dựng hệ thống RAG với LangChain và OpenAI

Mục tiêu

- Hiểu cách xử lý dữ liệu từ hệ thống tìm kiếm thông tin (retrieval).
- Hợp nhất dữ liệu đầu ra từ retrieval để sử dụng trong bước sinh văn bản (generation).
- Tạo prompt đơn giản để đưa vào mô hình sinh của OpenAI.
- Tích hợp LangChain để gọi API OpenAI một cách tiện lợi.
- Đánh giá hiệu suất và giới hạn của mô hình.

1. Tổng quan về bước retrieval

Sau khi hệ thống retrieval trả về kết quả, bạn sẽ nhận được một danh sách các tài liệu (docs_files) tương ứng với truy vấn của bạn.

Cấu trúc dữ liệu

```
len(docs_files)  # số lượng kết quả
docs_files[0]    # (page_content, score)
```

- `docs_files` là danh sách các tuple.
- Mỗi tuple gồm:
  - `page_content`: nội dung văn bản.
  - `score`: điểm số đánh giá mức độ liên quan.

2. Hợp nhất văn bản để dùng trong bước generation

Trước khi sinh câu trả lời, chúng ta cần ghép các `page_content` thành một khối văn bản lớn:

```
context_text = "\n\n".join([doc.page_content for doc, score in docs_files])
```

Mục đích:

- Tạo bối cảnh thống nhất cho mô hình sinh văn bản.
- Dễ dàng truyền vào prompt.

3. Tạo prompt đơn giản cho mô hình sinh

```
prompt = f"""Based on this context:

{context_text}

Please answer this question:
{query}

If you don’t know the answer, just say you don’t know."""
```

Lưu ý:

- Luôn khuyến khích mô hình trả lời "không biết" nếu không có thông tin.
- Tránh hiện tượng "hallucination" (mô hình bịa ra thông tin).

4. Gọi OpenAI API thông qua LangChain

```
from langchain.chat_models import ChatOpenAI

model = ChatOpenAI(
    openai_api_key=API_KEY,
    model_name="gpt-4o",
    temperature=0
)
```

- Sử dụng GPT-4o cho hiệu suất tốt hơn.
- `temperature = 0` để đảm bảo tính chính xác, không sáng tạo.

5. Thực thi truy vấn và hiển thị kết quả

```
response_text = model.invoke(prompt).content
display(Markdown(response_text))
```

- Gọi mô hình để sinh câu trả lời dựa trên `prompt`.
- Hiển thị dưới dạng Markdown để dễ đọc.

6. Phân tích và đánh giá kết quả

- Kết quả chưa hoàn hảo: trả về "no comment" hoặc thông tin không đầy đủ.
- Lý do:
  - Prompt chưa tối ưu.
  - Dữ liệu quá nhiều (2000 tokens x 5 docs).
  - Truy vấn chưa rõ ràng ("Give me my worst reviews with comments").

7. Hướng phát triển tiếp theo

- Tối ưu prompt để rõ ràng hơn (VD: yêu cầu kèm comment cụ thể).
- Giới hạn context hoặc lọc dữ liệu trước khi truyền vào mô hình.
- Tạo các **hàm tái sử dụng** cho quá trình merge, prompt, gọi API.

8. Kết luận

Dù chưa tối ưu, nhưng bạn đã hoàn tất một pipeline RAG đơn giản:

1. **Tìm kiếm thông tin (Retrieval)**: trả về các đoạn văn bản liên quan.
2. **Ghép context**: tạo ngữ cảnh thống nhất.
3. **Sinh văn bản (Generation)**: mô hình trả lời dựa vào context và truy vấn.

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft