RAG pipeline with MongoDB - Practice 6

Data	https://archive.org/download/AsimovTheFoundation/Asimov_the_foundation.pdf

Data Management and Visualization *- Politecnico di Torino*

Create and run a local RAG pipeline from scratch

The goal of this Lab is to build a RAG (Retrieval Augmented Generation) pipeline from scratch and have it run on a Colab Notebook.

Specifically, we will open a PDF file, ask questions (queries) and have them answered by a Large Language Model (LLM).

There are frameworks that replicate this kind of workflow, including LlamaIndex and LangChain, however, the goal of building from scratch is to be able to inspect and customize all the parts.

This document complements the operative steps you can find in the following Notebook:

https://colab.research.google.com/drive/1T4VgKpgTZ4hAAAhgBJmjJBMpssfMuR5H?usp=sharing

0. What is RAG?

RAG stands for **Retrieval Augmented Generation**.

It was introduced in the paper https://arxiv.org/abs/2005.11401

Each step can be roughly broken down to:

- **Retrieval** - Seeking relevant information from a source given a query. For example, getting relevant passages of Wikipedia text from a database given a question.
- **Augmented** - Using the relevant retrieved information to modify an input to a generative model (e.g. an LLM).
- **Generation** - Generating an output given an input. For example, in the case of an LLM, generating a passage of text given an input prompt.

Why RAG? The main goal of RAG is to improve the generation outptus of LLMs by:

1. Preventing hallucinations by adding relevant context to the input of an LLM.
2. Work with custom data sources the LLM may not have been trained on.

We're going to build RAG pipeline which enables us to chat with a PDF document, specifically an open-source Asimov's Foundation Novel Book, ~500 pages long.

We'll write the code to enable the following steps:

1. **Document Processing**
    a. Open a PDF document (you could use almost any PDF here).
    b. Format the text of the PDF textbook ready for an embedding model (this process is known as **text chunking**).
    c. Embed all of the chunks of text in the textbook and turn them into numerical representation which we can store for later.
2. **Vector Search and answer generation**
    a. Build a retrieval system that uses vector search to find relevant chunks of text based on a query.
    b. Create a prompt that incorporates the retrieved pieces of text.
    c. Generate an answer to a query based on passages from the textbook.

1. Document processing

The text processing text has been already provided for you. After downloading the sample PDF, the script reads it through the `open_and_read_pdf_by_outline` function which extracts text by chapters from a document's outline/table of contents. This is done to easily process big documents.

Text formatting occurs via the `text_formatter` function that cleans the text by removing newlines and extra spaces. You can customize this by adding additional text cleaning operations if needed.

For text chunking we first **separate the text into sentences**, in order not to have chunks with non-closed sentences. Then, we split the text into approximately fixed-size chunks of `chunk_size` characters, which are then extended by a certain number of extra sentences as padding. This operation add redundancy between chunks, but it **reduces possible information loss** of the chunking operation. Both chunking size and padding can be adjusted to fit the further embedding model's limitations.

After connecting to your personal MongoDB cluster, we can create a Collection and store our chunks as documents.

⚠️ **Warning:** This is still a document-based Database. For querying the text as a in a Vector database we first need to *embed* them.

> ***TODO:*** Following the notebook, we can inspect our database and get useful insights for the further steps.

Moving to the **embedding phase**, we are going to use `all-mpnet-base-v2` as embedding model. It has an input capacity of 384.

> ***Question:*** Are the chunks we did adequate for this model? If it is not the case, we can repeat the above process to fix them.

Finally, let's store the chunks to MongoDB as a **Vector Database**.

While humans understand text, machines understand numbers best. The most powerful thing about modern embeddings is that they are *learned* representations.

Meaning rather than directly mapping words/tokens/characters to numbers directly (e.g. `{"a": 0, "b": 1, "c": 3...}`), the numerical representation of tokens is learned by going through large corpuses of text and figuring out how different tokens relate to each other.

Ideally, embeddings of text will mean that similar meaning texts have similar numerical representation. Our goal is to turn each of our chunks into a numerical representation (an embedding vector, where a vector is a sequence of numbers arranged in order). Once our text samples are in embedding vectors, us humans will no longer be able to understand them.

To do so, we'll use the `sentence-transformers` library which contains many pre-trained embedding models. Specifically, we'll get the `all-mpnet-base-v2` model (you can see the model's intended use on the Hugging Face model card).

2. Vector Search

A vector database is a specialized type of database designed to efficiently store, retrieve, and manage vector embeddings. Unlike traditional databases that use exact matching for queries, vector databases operate on the principle of **similarity search**, using algorithms to find the most similar vectors to a query vector. The main building blocks of a vector database include:

- The **vector store** which contains the embedded representations of documents or chunks of text. MongoDB will be our vector store.
- The **similarity search functionality** which finds the most relevant vectors using distance metrics like cosine similarity.

- The **indexing mechanism** that organizes vectors to enable efficient similarity search (using algorithms like HNSW, IVF, etc.). We will get into this later on.

For our RAG pipeline, we'll use MongoDB with vector search capabilities to store our embedded text chunks and perform semantic searches when answering questions about the text.

The distance metrics we will use is Cosine Similarity:

- Range: -1 (opposite) to +1 (identical)
- Higher values = most similar vectors

Example

- `"AI is amazing!"` vs. `"AI is incredible!"` → Cosine Similarity ≈ 0.95 (high similarity).
- `"AI is amazing!"` vs. `"I love ice cream!"` → Cosine Similarity ≈ 0.05 (low similarity).

Before implementing a real Vector Database with all the optimizations, we will simulate the steps with one examples. The steps to follow are:

- Build the embedding of a user query (e.g. *"What is the Galactic Empire?"*)
- Fetch all embeddings from our collection and compare them to a user query using cosine similarity
- Return the top-k most similar chunks

Going on through the notebook, you will understand how to optimize these steps into our MongoDB database.

3. Prompt construction and answer generation

A prompt is a set of instructions given to a language model to guide its response generation. In a RAG context, prompts are designed to effectively incorporate retrieved information from a knowledge base to produce accurate and relevant answers.

When designing prompts for our RAG pipeline, consider these key principles:

- **Context integration:** Include retrieved text chunks from our vector search in the prompt, providing the model with the necessary information to answer accurately.
- **Clear instruction:** Explicitly tell the model to base its answers on the provided context and not to use information outside of it.
- **Source attribution:** Ask the model to cite or reference the specific parts of the text it's drawing information from.

- **Handling uncertainty:** Include instructions on what to do when the retrieved context doesn't contain enough information to answer the query (e.g., admit knowledge gaps).

A typical RAG prompt structure might look like:

```
prompt = f"""
Answer the following question based ONLY on the provided context.
If you cannot answer from the context, state "I don't have enough information."

Context:
{retrieved_context}

Question: {user_question}

Answer:
"""
```

This structure ensures that the LLM stays grounded in the retrieved information, reducing hallucinations and hopefully improving answer accuracy.

Finally, we can load a **Large Language Model** (our Generator) and start asking questions! To do so, we will use https://huggingface.co/, which is comprehensive platform for AI models and resources that provides access to thousands of pre-trained models, datasets, and tools for machine learning development and deployment.

You can follow the instruction to setup the LLM. For simplicity we will access it from an InferenceClient (https://huggingface.co/docs/huggingface_hub/en/guides/inference). You can also experiment other option such as running it locally using available GPUs or even use premium APIs from OpenIA, Google Gemini, etc.

Our RAG workflow is completed!

We've now officially got a way to Retrieve, Augment and Generate answers based on a source.

For now we can verify our answers manually by reading them and reading through the textbook. But if you want to put this into a production system, it'd be a good idea to have some kind of evaluation on how well our pipeline works.

For example, you could use another LLM to rate the answers returned by our LLM and then use those ratings as a proxy evaluation.

However, We'll leave this and a few more interesting ideas as extensions.

4. Extensions

- Try another embedding model (you can choose from the following leaderboard https://huggingface.co/spaces/mteb/leaderboard)
- See the following prompt engineering resources for more prompting techniques (e.g. https://www.promptingguide.ai/).
- Try another LLM... (take inspiration from here, https://huggingface.co/open-llm-leaderboard... Mind the model size!).
- Our example only focuses on text from a PDF, however, we could extend it to include figures and images. How does the pipeline change?
- Evaluate the answers: could use another LLM to rate our answers. A library for full RAG evaluation is https://docs.ragas.io/en/stable/.