Back to Articles

Code a simple RAG from scratch

Community Article Published October 29, 2024

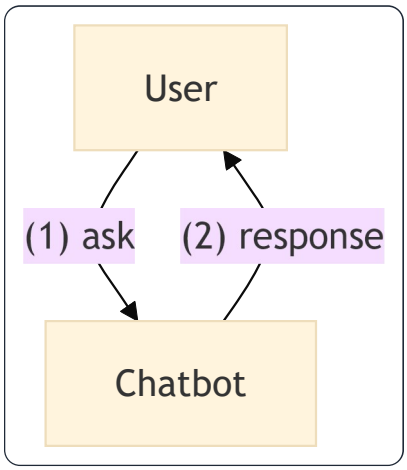
ngxson
Xuan Son NGUYEN

Recently, **Retrieval-Augmented Generation (RAG)** has emerged as a powerful paradigm in the field of AI and Large Language Models (LLMs). RAG combines information retrieval with text generation to enhance language models' performance by incorporating external knowledge sources. This approach has shown promising results in various applications, such as question answering, dialogue systems, and content generation.

In this blog post, we'll explore RAG and build a simple RAG system from scratch using **Python** and **ollama**. This project will help you understand the key components of RAG systems and how they can be implemented using fundamental programming concepts.

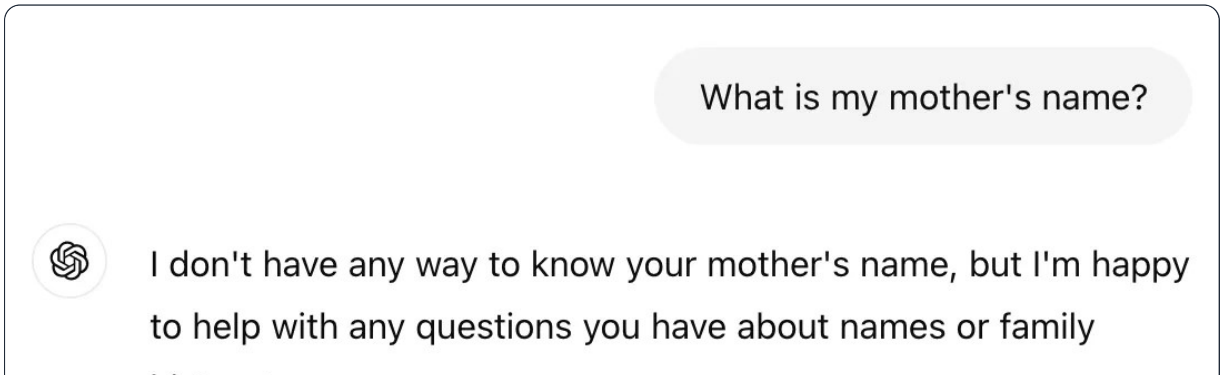
What is RAG

To begin, let's examine a simple chatbot system **without RAG**:



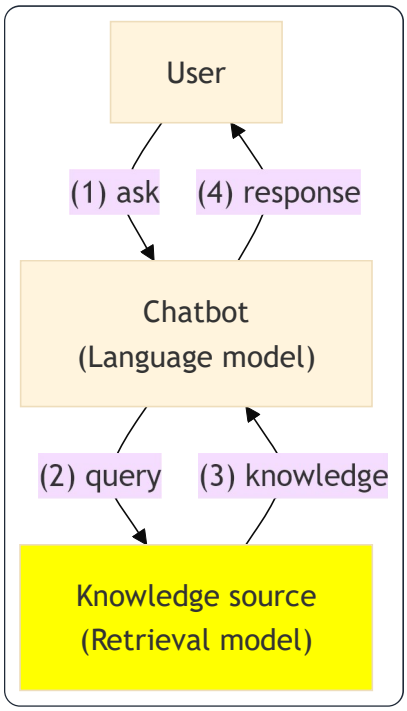
While the chatbot can respond to common questions based on its training dataset, it may lack access to the most up-to-date or domain-specific knowledge.

A real-world example would be asking ChatGPT "What is my mother's name?". ChatGPT cannot answer this question because it doesn't have access to external knowledge, such as your family members' information.



history!

To address this limitation, we need to provide external knowledge to the model (in this example, a list of family members' names):



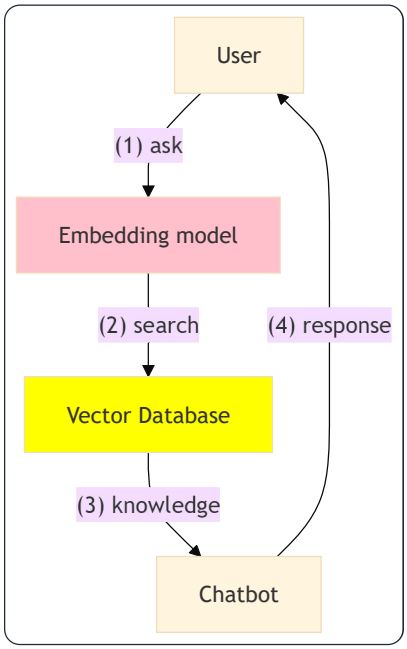
A RAG system consists of two key components:

- A **retrieval model** that fetches relevant information from an external knowledge source, which could be a database, search engine, or any other information repository.
- A **language model** that generates responses based on the retrieved knowledge.

There are several ways to implement RAG, including Graph RAG, Hybrid RAG, and Hierarchical RAG, which we'll discuss at the end of this post.

Simple RAG

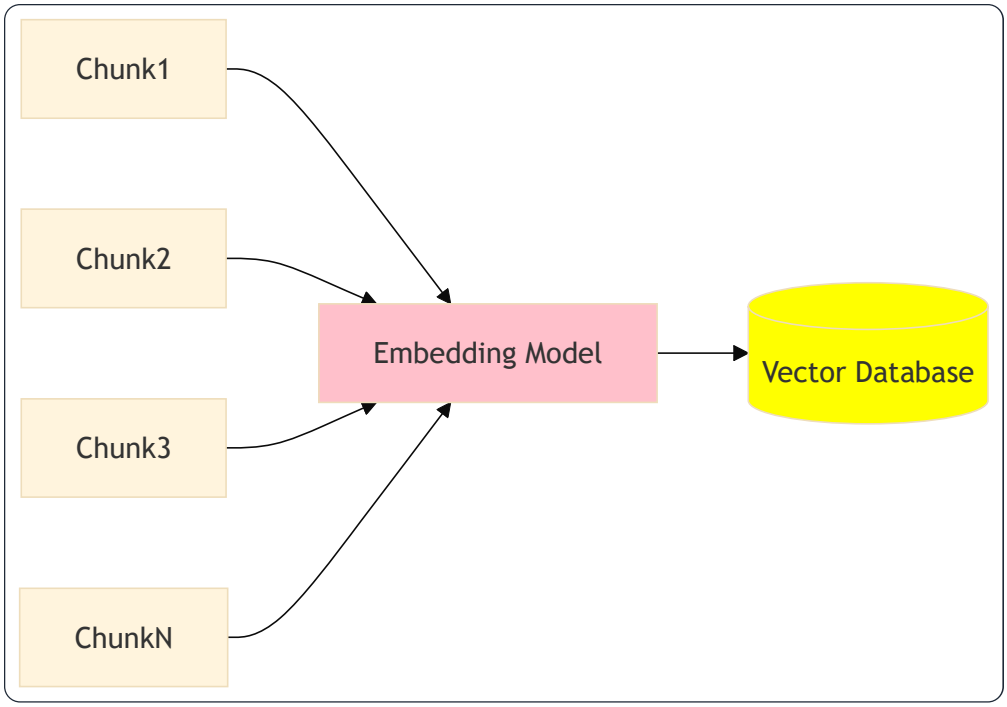
Let's create a simple RAG system that retrieves information from a predefined dataset and generates responses based on the retrieved knowledge. The system will comprise the following components:



1. **Embedding model**: A pre-trained language model that converts input text into embeddings - vector representations that capture semantic meaning. These vectors will be used to search for relevant information in the dataset.
2. **Vector database**: A storage system for knowledge and its corresponding embedding vectors. While there are many vector database technologies like Qdrant, Pinecone, and pgvector, we'll implement a simple in-memory database from scratch.
3. **Chatbot**: A language model that generates responses based on retrieved knowledge. This can be any language model, such as Llama, Gemma, or GPT.

Indexing phase

The indexing phase is the first step in creating a RAG system. It involves breaking the dataset (or documents) into small **chunks** and calculating a vector representation for each chunk that can be efficiently searched during generation.



The size of each chunk can vary depending on the dataset and the application. For example, in a document retrieval system, each chunk can be a paragraph or a sentence. In a dialogue system, each chunk can be a conversation turn.

After the indexing phrase, each chunk with its corresponding embedding vector will be stored in the vector database. Here is an example of how the vector database might look like after indexing:

Chunk	Embedding Vector
Italy and France produce over 40% of all wine in the world.	\[0.1, 0.04, -0.34, 0.21, ...\]
The Taj Mahal in India is made entirely out of marble.	\[-0.12, 0.03, 0.9, -0.1, ...\]
90% of the world's fresh water is in Antarctica.	\[-0.02, 0.6, -0.54, 0.03, ...\]
...	...

The embedding vectors can be later used to retrieve relevant information based on a given query. Think of it as SQL `WHERE` clause, but instead of querying by exact text matching, we can now query a set of chunks based on their vector representations.

To compare the similarity between two vectors, we can use cosine similarity, Euclidean distance, or other distance metrics. In this example, we will use cosine similarity. Here is the formula for cosine similarity between two vectors A and B:

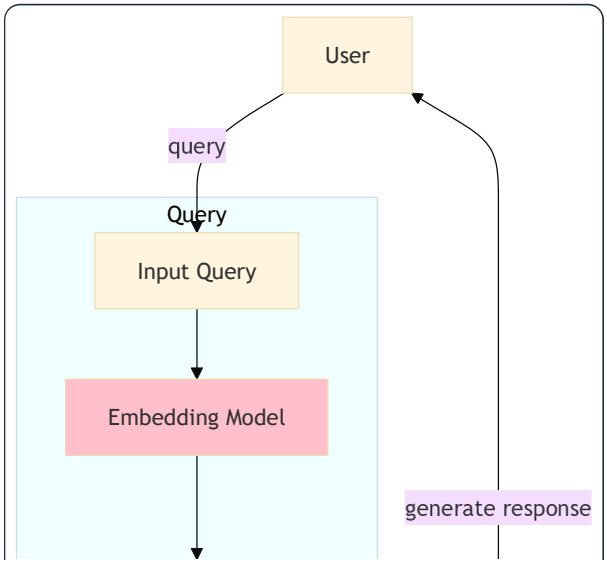
$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sum_{i=1}^{n} B_i^2}.$$

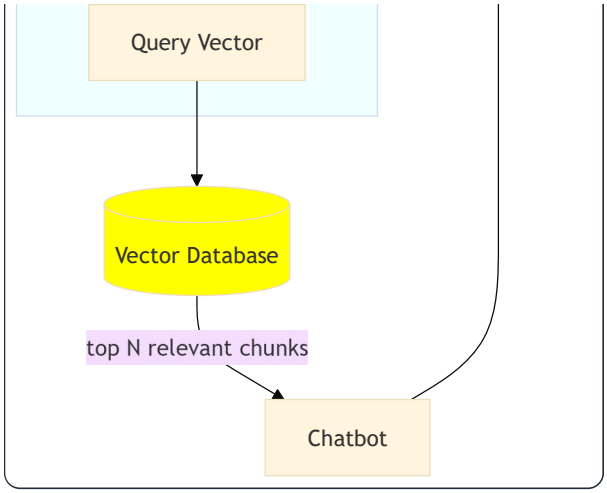
Don't worry if you are not familiar with the formula above, we will implement it in the next section.

Retrieval phrase

In the diagram below, we will take an example of a given `Input Query` from `User`. We then calculate the `Query Vector` to represent the query, and compare it against the vectors in the database to find the most relevant chunks.

The result returned by The `Vector Database` will contains top N most relevant chunks to the query. These chunks will be used by the `Chatbot` to generate a response.





Let's code it

In this example, we will write a simple **Python** implement of RAG.

To run the models, we will use ollama, a command line tool that allows you to run models from Hugging Face. With ollama, you **don't need** to have access to a server or cloud service to run the models. You can run the models directly on your computer.

For the models, let's use the following:

- **Embedding model**: hf.co/CompendiumLabs/bge-base-en-v1.5-gguf
- **Language model**: hf.co/bartowski/Llama-3.2-1B-Instruct-GGUF

And for the dataset, we will use a simple list of facts about cat. Each fact will be considered as a chunk in the indexing phrase.

Download ollama and models

First, let's start by installing ollama from project's website: ollama.com

After installed, open a terminal and run the following command to download the

required models:

```
ollama pull hf.co/CompendiumLabs/bge-base-en-v1.5-gguf
ollama pull hf.co/bartowski/Llama-3.2-1B-Instruct-GGUF
```

If you see the following output, it means the models are successfully downloaded:

```
pulling manifest
...
verifying sha256 digest
writing manifest
success
```

Before continuing, to use `ollama` in python, let's also install the `ollama` package:

```
pip install ollama
```

Loading the dataset

Next, create a Python script and load the dataset into memory. The dataset contains a list of cat facts that will be used as chunks in the indexing phrase.

You can download the example dataset from here. Here is an example code to load the dataset:

```
dataset = []
with open('cat-facts.txt', 'r') as file:
  dataset = file.readlines()
  print(f'Loaded {len(dataset)} entries')
```

Implement the vector database

Now, let's implement the vector database.

We will use the embedding model from `ollama` to convert each chunk into an embedding vector, then store the chunk and its corresponding vector in a list.

Here is an example function to calculate the embedding vector for a given text:

```
import ollama

EMBEDDING_MODEL = 'hf.co/CompendiumLabs/bge-base-en-v1.5-gguf'
LANGUAGE_MODEL = 'hf.co/bartowski/Llama-3.2-1B-Instruct-GGUF'

# Each element in the VECTOR_DB will be a tuple (chunk, embedding)
# The embedding is a list of floats, for example: [0.1, 0.04, -0.34, 0
VECTOR_DB = []

def add_chunk_to_database(chunk):
  embedding = ollama.embed(model=EMBEDDING_MODEL, input=chunk)['embedd
  VECTOR_DB.append((chunk, embedding))
```

In this example, we will consider each line in the dataset as a chunk for simplicity.

```
for i, chunk in enumerate(dataset):
  add_chunk_to_database(chunk)
  print(f'Added chunk {i+1}/{len(dataset)} to the database')
```

Implement the retrieval function

Next, let's implement the retrieval function that takes a query and returns the top N most relevant chunks based on cosine similarity. We can imagine that the higher the cosine similarity between the two vectors, the "closer" they are in the vector space.

This means they are more similar in terms of meaning.

Here is an example function to calculate the cosine similarity between two vectors:

```
def cosine_similarity(a, b):
  dot_product = sum([x * y for x, y in zip(a, b)])
  norm_a = sum([x ** 2 for x in a]) ** 0.5
  norm_b = sum([x ** 2 for x in b]) ** 0.5
  return dot_product / (norm_a * norm_b)
```

Now, let's implement the retrieval function:

```
def retrieve(query, top_n=3):
  query_embedding = ollama.embed(model=EMBEDDING_MODEL, input=chunk)['
  # temporary list to store (chunk, similarity) pairs
  similarities = []
  for chunk, embedding in VECTOR_DB:
    similarity = cosine_similarity(query_embedding, embedding)
    similarities.append((chunk, similarity))
  # sort by similarity in descending order, because higher similarity
  similarities.sort(key=lambda x: x[1], reverse=True)
  # finally, return the top N most relevant chunks
  return similarities[:top_n]
```

Generation phrase

In this phrase, the chatbot will generate a response based on the retrieved knowledge from the step above. This is done by simply add the chunks into the prompt that will be taken as input for the chatbot.

For example, a prompt can be constructed as follows:

```
input_query = input('Ask me a question: ')
```

```
retrieved_knowledge = retrieve(input_query)

print('Retrieved knowledge:')
for chunk, similarity in retrieved_knowledge:
  print(f' - (similarity: {similarity:.2f}) {chunk}')

instruction_prompt = f'''You are a helpful chatbot.
Use only the following pieces of context to answer the question. Don't
{'\n'.join([f' - {chunk}' for chunk, similarity in retrieved_knowledge
'''
```

We then use the `ollama` to generate the response. In this example, we will use `instruction_prompt` as system message:

```
stream = ollama.chat(
  model=LANGUAGE_MODEL,
  messages=[
    {'role': 'system', 'content': instruction_prompt},
    {'role': 'user', 'content': input_query},
  ],
  stream=True,
)

# print the response from the chatbot in real-time
print('Chatbot response:')
for chunk in stream:
  print(chunk['message']['content'], end='', flush=True)
```

Putting it all together

You can find the final code in this file. To run the code, save it to a file named `demo.py` and run the following command:

```
python demo.py
```

You can now ask the chatbot questions, and it will generate responses based on the retrieved knowledge from the dataset.

```
Ask me a question: tell me about cat speed
Retrieved chunks: ...
Chatbot response:
According to the given context, cats can travel at approximately 31 mp
```

Rooms for improvement

So far, we have implemented a simple RAG system using a small dataset. However, there are still many limitations:

- If the question covers **multiple topics** at the same time, the system may not be able to provide a good answer. This is because the system only retrieves chunks based on the similarity of the query to the chunks, without considering the context of the query.
  The solution could be to have the chatbot to write its own query based on the user's input, then retrieve the knowledge based on the generated query. We can also use multiple queries to retrieve more relevant information.
- The top N results are returned based on the cosine similarity. This may not always give the best results, especially when each chunks contains a lot of information.
  To address this issue, we can use a reranking model to **re-rank the retrieved chunks** based on their relevance to the query.
- The database is stored in memory, which may not be scalable for large datasets. We can use a more efficient vector database such as Qdrant, Pinecone, pgvector

- We currently consider each sentence to be a chunk. For more complex tasks, we may need to use more sophisticated techniques to **break down the dataset into smaller chunks**. We can also pre-process each chunk before adding them to the database.
- The language model used in this example is a simple one which only has 1B parameters. For more complex tasks, we may need to use a larger language model.

Other types of RAG

In practice, there are many ways to implement RAG systems. Here are some common types of RAG systems:

- **Graph RAG**: In this type of RAG, the knowledge source is represented as a graph, where nodes are entities and edges are relationships between entities. The language model can traverse the graph to retrieve relevant information. There are many active researches on this type of RAG. Here is a collection of papers on Graph RAG.
- **Hybrid RAG**: a type of RAG that combines Knowledge Graphs (KGs) and vector database techniques to improve question-answering systems. To know more, you can read the paper here.
- **Modular RAG**: a type of RAG that goes beyond the basic "retrieve-then-generate" process, employing routing, scheduling, and fusion mechanisms to create a flexible and reconfigurable framework. This modular design allows for various RAG patterns (linear, conditional, branching, and looping), enabling more sophisticated and adaptable knowledge-intensive applications. To know more, you can read the paper here.

For other types of RAG, you can refer to the this post by Rajeev Sharma.

Conclusion

RAG represents a significant advancement in making language models more knowledgeable and accurate. By implementing a simple RAG system from scratch, we've explored the fundamental concepts of embedding, retrieval, and generation. While our implementation is basic, it demonstrates the core principles that power more sophisticated RAG systems used in production environments.

The possibilities for extending and improving RAG systems are vast, from implementing more efficient vector databases to exploring advanced architectures like Graph RAG and Hybrid RAG. As the field continues to evolve, RAG remains a crucial technique for enhancing AI systems with external knowledge while maintaining their generative capabilities.

References

- https://arxiv.org/abs/2005.11401
- https://aws.amazon.com/what-is/retrieval-augmented-generation/
- https://github.com/varunvasudeva1/llm-server-docs
- https://github.com/ollama/ollama/blob/main/docs
- https://github.com/ollama/ollama-python
- https://www.pinecone.io/learn/series/rag/rerankers/
- https://arxiv.org/html/2407.21059v1
- https://newsletter.armand.so/p/comprehensive-guide-rag-implementations

---

TOS

Privacy

About

Jobs

Website

Models

Datasets

Spaces

Pricing

Docs